Programação e Sistemas de Informação

Módulo 13

Técnicas de Modelação de Dados

Aula 13

Preparação para a semana

1) Abrir uma consola // Git Bash/PowerShell
2) cd MinhaPastaDePSI // Nome à vossa escolha
3) mkdir Semana13
4) cd Semana13
5) dotnet new sln // Cria solução
6) git add .
7) git commit -m “Adicionar solução para semana 13”
8) git push
9) Visual Studio Code → Open Folder → Semana13

Cábula para comandos dotnet

- dotnet new console -n NomeDoProjeto
  - Cria novo projeto chamado “NomeDoProjeto”
- dotnet sln add NomeDoProjeto
  - Adiciona projeto “NomeDoProjeto” à solução existente
- dotnet run -p NomeDoProjeto
  - Compila e executa projeto “NomeDoProjeto”

Cábula para projetos do dotnet 6.0

- dotnet new -i Classic.Console.Templates
  - Instala *templates* para versões anteriores do dotnet
  - Apenas é preciso correr este comando uma vez por sistema
- dotnet new console-classic --nrt=false --langVersion 8.0 -n NomeDoProjeto
  - Cria projeto com template de *framework* anterior à versão 6.0

Conteúdos

- ➔ Interfaces
- ➔ Coleções

Interfaces no mundo real

- Conceitos que definem o que um objeto tem e pode fazer
- Exemplos:
  - Combustível
    - Pode arder, tem energia produzida
    - Gasolina, carvão, papel, madeira
  - Instrumento de desenho
    - Pode desenhar, tem cor
    - Pincel, carvão, caneta
  - Superfície de desenho
    - Pode ser desenhada, tem área
    - Papel, madeira, cartolina

Interfaces em programação por objetos

- Definem de forma conceptual o que um objeto tem e pode fazer
- Oferecem uma forma possível de interpretar objetos
  - Exemplo: um array com objetos do tipo **Gasolina**, **Madeira**, **Papel**
    - Todos estes objetos podem ser interpretados como **Combustivel**
- São um contrato a que as classes têm de obedecer
  - Exemplo: classe **Papel** implementa a interface **SuperficieDeDesenho**
    - Esta contém a propriedade **Area** e o método **SerDesenhada()**
    - Classe **Papel** é obrigada a usar esta propriedade e este método

Criar uma interface em C#

```
public interface ICombustivel
{
    float Energia { get; }
    void Queimar();
}
```

- Métodos e propriedades não têm corpo
    - São **public** e **abstract** por omissão
- É boa prática começar o nome com **I** (i maiúsculo)

Usar uma interface em C#

- Relação de classes com interfaces é de **implementação**
  - Classes **implementam** tantas interfaces quanto quiserem
  - Se classes estenderem uma classe base, esta tem de aparecer primeiro
  - Classes indicam interfaces implementadas após **:** (igual à herança)

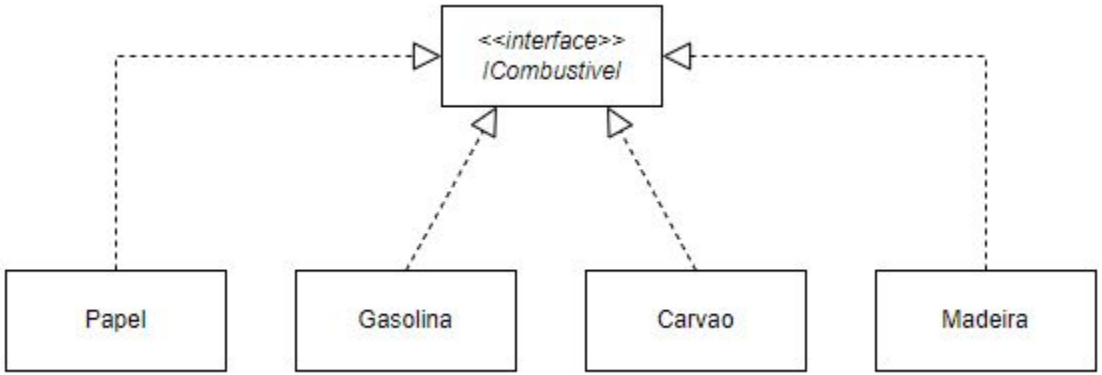
```
public class Papel : ICombustivel
{
    public float Energia { get { return 5f; } }
    public void Queimar()
    {
        Console.WriteLine(“Papel está a queimar”);
    }
}
```

Interfaces - exemplo de uso

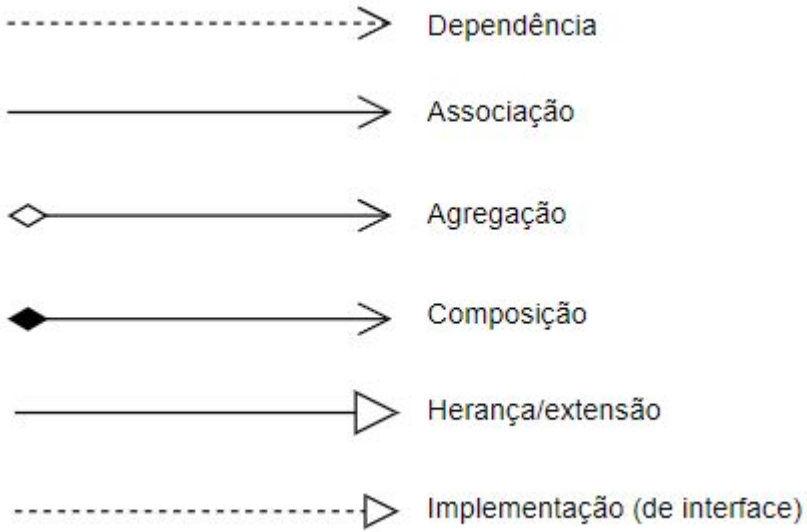
```
ICombustivel variosCombustiveis = new ICombustivel[4];

variosCombustiveis[0] = new Papel();
variosCombustiveis[0] = new Gasolina();
variosCombustiveis[0] = new Carvao();
variosCombustiveis[0] = new Madeira();

foreach (ICombustivel combustivel in variosCombustiveis)
{
    Console.WriteLine(combustivel.Energia);
}
```

Interfaces - representação em UML

- Antes do nome, incluir **<<*interface*>>**
  - Em adição, o nome pode ser itálico para reforçar o seu caráter abstrato
- Para representar relação de implementação de interface:
  - Usar uma linha tracejada com triângulo fechado



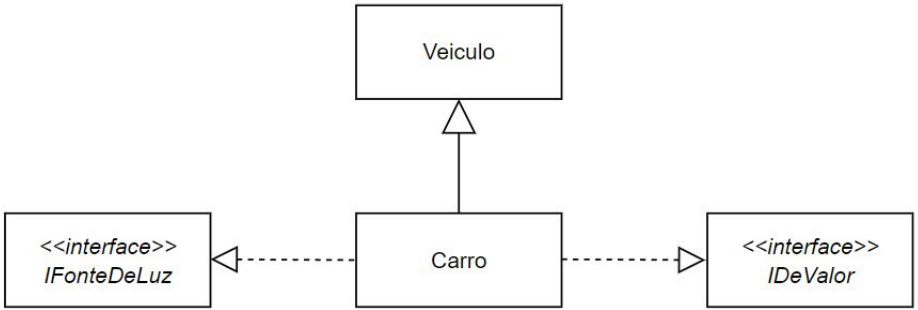
Relações UML em revisão



Herança múltipla

- Suportada em algumas linguagens // Exemplo: C++
- C# não suporta herança múltipla
  - Regra geral, traz vários problemas e complica o código
  - Tornar-se-ia difícil perceber qual a classe base onde estão os métodos e propriedades
- Interfaces resolvem este problema
  - Indicam membros a implementar, mas não os implementam
  - Só existe uma classe base

```
public class Carro : Veiculo, IFonteDeLuz, IDeValor
{
    public override void Acelerar() { … }
    public void GerarLuz() { … }
    public float GetCusto() { … }
}
```



Exercício 1

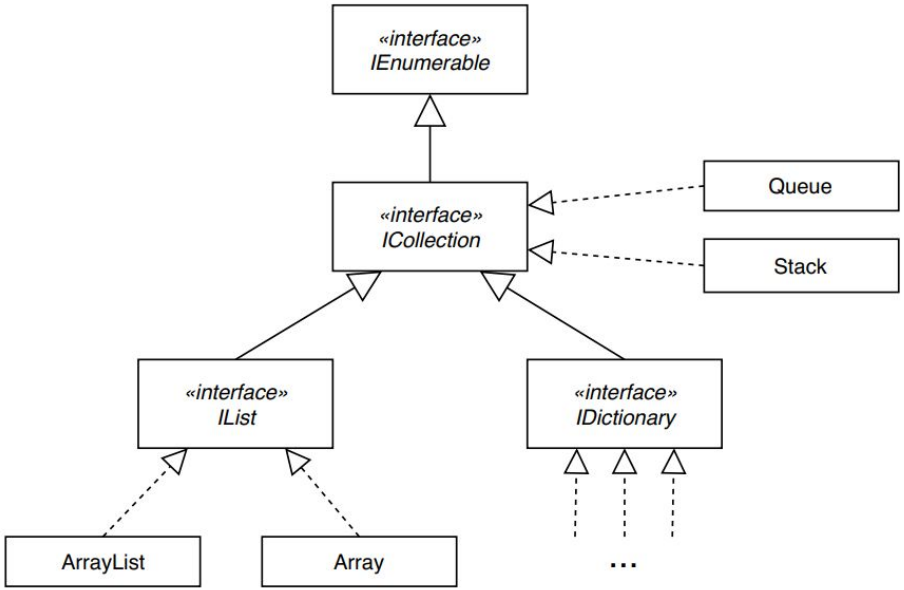
- ➔ Antes de começar a programar, ler o enunciado e desenhar o diagrama UML das classes
- ➔ Criar projeto **Animais** na solução **Semana13**
  - ◆ Criar as classes **Animal** e **Gato**, tal como se apresentam no *slide* 19 da aula 8
  - ◆ Criar as classes **Cao**, **Morcego** e **Abelha**, seguindo a mesma lógica da classe **Gato**
  - ◆ Criar a interface **IMamifero**, com a propriedade **NumeroDeMamilos** (*read-only*, tipo int)
    - Apenas classes que representam animais mamíferos devem implementar esta interface
    - Propriedade deve corresponder ao número de mamilos que esses animais têm
  - ◆ Criar a interface **IVoador**, com a propriedade **NumeroDeAsas** (*read-only*, tipo int)
    - Apenas classes que representam animais voadores devem implementar esta interface
    - Propriedade deve corresponder ao número de asas que esses animais têm
  - ◆ Na classe **Program**, método **Main()**, testar classes anteriores:
    - Criar *array* com 10 animais aleatórios e apresentar, para cada um deles:
      - ○ A frase devolvida pelo método **Som()**
      - ○ Caso seja mamífero, o número de mamilos
      - ○ Caso seja voador, o número de asas
- ➔ Fazer vários *commits* e, no fim do exercício, *push* para o repositório remoto

Conteúdos

- ➔ Interfaces
- ➔ Coleções

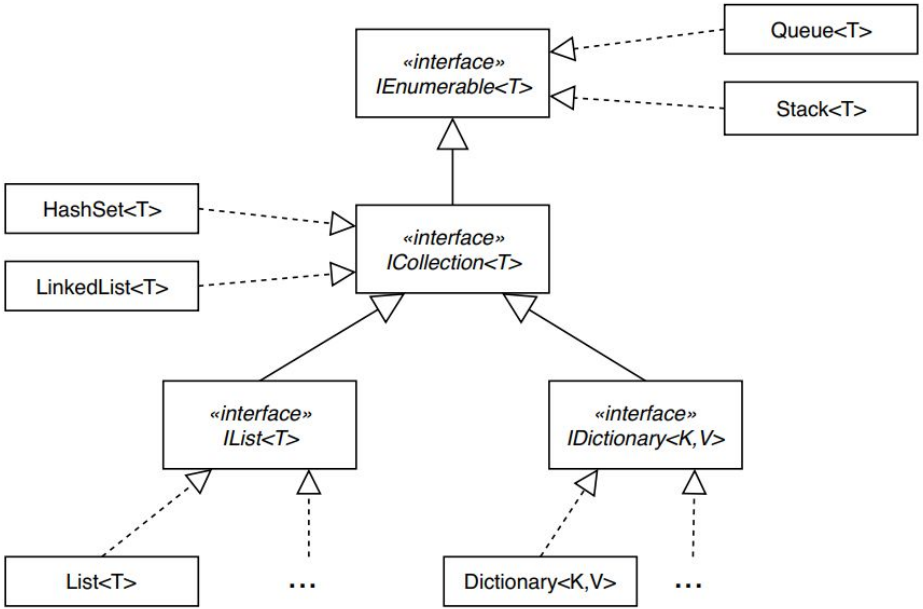
Coleções não-genéricas

- *Namespace* **System.Collections**



Principais coleções genéricas em C#

- *Namespace* **System.Collections.Generic**



Coleções genéricas em C#

- Tipo de objetos a guardar é indicado na declaração
  - Lista de *strings*: **List<string> strings = new List<string>();**
- Só aceitam objetos do tipo especificado
  - **strings.Add(“Uma string”);** // Válido
  - **strings.Add(123);** // Inválido
- Devolvem diretamente objetos do tipo especificado
  - **string s = strings[3];**

Coleções genéricas mais comuns em C#

- **List<T>**
  - Lista indexada de objetos
  - Métodos comuns: **Add()**, **Insert()**, **Remove()**, **Sort()**
- **Stack<T>**
  - Pilha: último a entrar, primeiro a sair (LIFO)
  - Métodos comuns: **Push()**, **Pop()**, **Peek()**
- **Queue<T>**
  - Fila: primeiro a entrar, primeiro a sair (FIFO)
  - Métodos comuns: **Enqueue()**, **Dequeue()**, **Peek()**
- **HashSet<T>**
  - Conjunto de objetos diferentes
  - Métodos comuns: **Add()**, **Contains()**, **IntersectWith()**, **UnionWith()**
- **Dictionary<K, V>**
  - Tabela de objetos indexados por chaves diferentes
  - Métodos comuns: **ContainsKey()**, **ContainsValue()**, **Add()**, **Remove()**

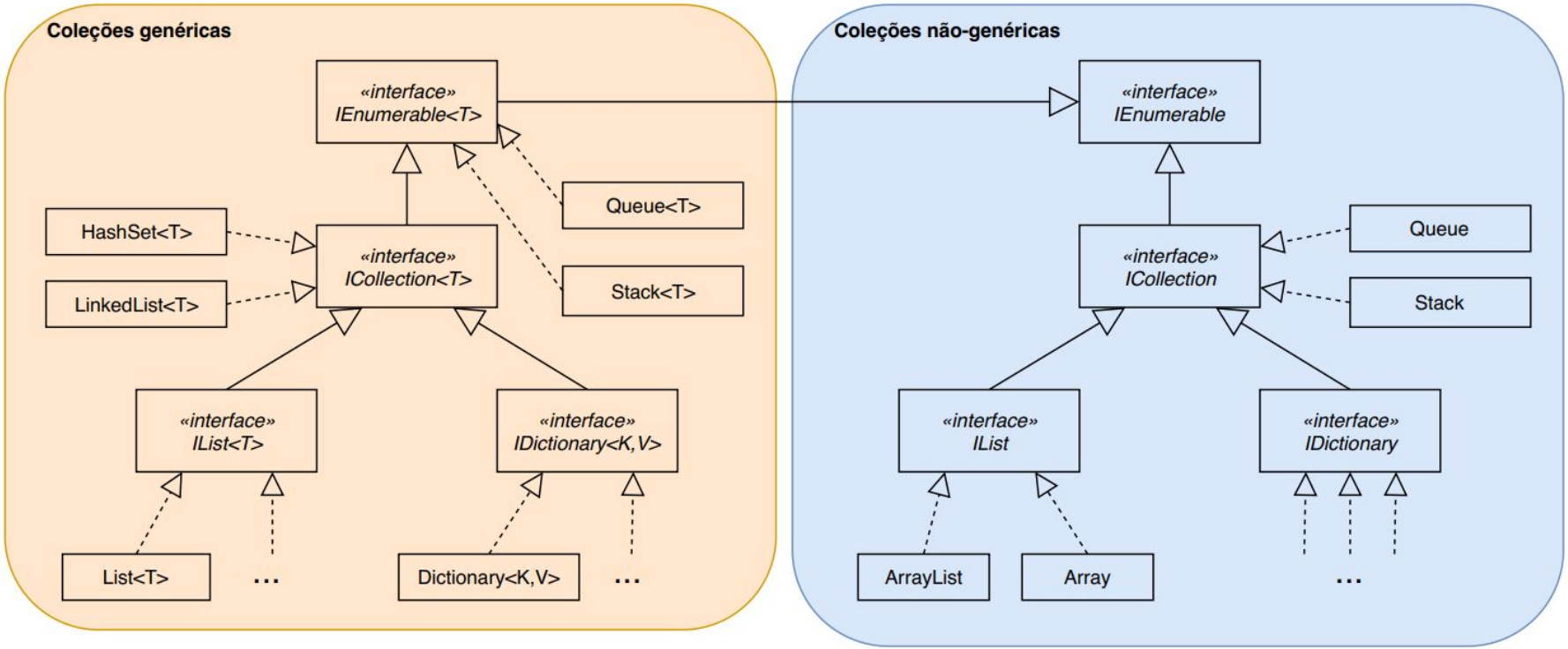
Sintaxe de inicialização de coleções

- Exemplo: **List<int> lista = new List<int> { 1, 2, 3, 4 };**
  - Para que uma classe possa ser inicializada assim:
    - Tem de implementar **IEnumerable<T>**
    - Tem de possuir um método **Add()**

A interface IEnumerable<T>

- **foreach** pode ser usado em instâncias de classes
  - Estas têm de implementar **IEnumerable** ou **IEnumerable<T>**
    - Coleções não-genéricas implementam **IEnumerable**
    - Coleções genéricas implementam **IEnumerable<T>**
    - **IEnumerable<T>** estende **IEnumerable**
- Podem-se percorrer os itens de qualquer classe que implemente esta interface
  - Exemplos: **string**, **array**, **List<T>**, **Queue<T>**

Organização das coleções em C#



Exercício 2

- ➔ Criar projeto **GestorAnimais** na solução **Semana13**
  - Criar classe **Animal** com as seguintes propriedades:
    - **Nome**, do tipo *string*, só de leitura
    - **Peso**, do tipo int
  - O programa deve gerar uma lista de animais, com as seguintes funcionalidades:
    - Inserir animal (nome e peso)
    - Listar todos os animais, indicando nome e peso
    - Listar animais com peso maior que valor indicado pelo utilizador
      - Criar o método **GetAnimaisComPesoMaiorQue()**
      - Este devolve uma coleção que implemente **IEnumerable<Animal>**
    - Devem existir dois animais por omissão
      - Estes são adicionados à lista no momento da criação da mesma
      - Usar sintaxe de inicialização de coleções
- ➔ Fazer vários *commits* e, no fim do exercício, *push* para o repositório remoto

Métodos que devolvem IEnumerable<T>

- No último exercício foi necessário criar uma coleção temporária
  - Isto apenas para iterar sobre a mesma
- C# permite devolver itens sem ter de criar uma coleção temporária
  - **yield return item;**

```
public class NumerosReais
{
    public IEnumerable<float> GetFloats()
    {
        yield return 3.2f;
        yield return 7.5f;
        yield return -1.6f;
    }
}
```

```
public static void Main()
{
    NumerosReais nr = new NumerosReais();
    foreach(float f in nr.GetFloats())
    {
        Console.WriteLine(f);
    }
}
```

Ciclos em métodos iteráveis e yield break

- Geralmente os métodos iteráveis contêm ciclos
  - **yield return** é usado para ir devolvendo itens
  - **yield break** pode ser usado para finalizar a devolução de itens

```
public IEnumerable<Jogador> GetCincoJogadoresComScoreMaiorQue(int i)
{
    int numeroDeJogadores = 0;
    foreach (Jogador j in jogadores) //  jogadores é uma variável de instância do tipo Jogador[]
    {
        if (j.Score > i)
        {
            numeroDeJogadores++;
            yield return j;
        }
        if (numeroDeJogadores >= 5) yield break;
    }
}
```

Exercício 3

- ➔ No projeto **GestorAnimais**:
  - ◆ Atualizar o método **GetAnimaisComPesoMaiorQue()**
    - Este deve agora ir devolvendo animais com **yield return** em vez de criar uma nova coleção
- ➔ Fazer vários *commits* e, no fim do exercício, *push* para o repositório remoto

A interface IComparable<T>

- Compara valores/instâncias para fins de ordenação
- Define um método para comparar uma instância com outra
  - **public int CompareTo(T other)**
  - Retorna 0 se instância e **other** não tiverem diferenças em termos de ordenação
  - Retorna < 0 se instância vier antes de **other**
  - Retorna > 0 se instância vier depois de **other**

```
Console.Write(“Ola”.CompareTo(“Ola”)); // Imprime 0
Console.Write(3.14f.CompareTo(25.6f)); // Imprime -1
 Console.Write(2.CompareTo(1)); // Imprime 1
```

IComparable<T> para ordenar instâncias numa coleção

- Pode-se usar qualquer valor para ordenar uma coleção
  - Exemplo: ordenar jogadores pelo seu *high score*:

```
public int CompareTo(Jogador other)
{
    if (other == null) return 1;
    return other.Score - this.Score
}
```

- Para ordenar, algumas coleções têm o método **Sort()**
  - Exemplo com uma lista de números inteiros:

```
List<int> lista = new List<int>() { 7, 5, 9, 3 };
lista.Sort();
foreach (int i in lista) Console.Write(i + " ");
```

Exercício 4

- No projeto **GestorAnimais**:
  - Atualizar a classe **Animal** de modo a implementar **IComparable<Animal>**
  - As listas de animais devem agora aparecer ordenadas por **Peso** (do maior para o menor)
- Fazer vários *commits* e, no fim do exercício, *push* para o repositório remoto