**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS**
Instituto de Informática e Ciências Exatas - Curso de Sistemas de Informação
Laboratório de Programação Orientada por Objeto - Professor: Paulo Amaral

# RELATÓRIO 3 (2/2015) - TRABALHO EM GRUPO

# Classes abstratas, Interfaces e Polimorfismo

## ESPECIFICAÇÕES

Deve ser **entregue** um texto com o **relatório completo** do trabalho que conste na capa (1ª. página) os **nomes completos de todos os componentes do grupo**. O número de componentes do grupo será especificado pelo professor durante a aula, não sendo aceitos trabalhos anônimos.

No **relatório** deve constar tudo que foi especificado no exercício e os c**ódigos-fonte de todos os programas** envolvidos nos exercícios**.**

Todos programas devem **imprimir o nome completo de todos os componentes do grupo na tela** e os **arquivos fontes dos programas, além de serem incluídos no texto do relatório, devem ser entregues em arquivos separados no SGA**, devendo conter o **cabeçalho completo** com entrada, saída, descrição e nome completo de todos os componentes do grupo (não sendo aceitos trabalhos sem os nomes. **Cópias grosseiras** serão desconsideradas, ou seja, a **nota será igual a 0 (zero).**

**Haverá apresentação oral dos trabalhos**

## 1. Métodos e classes abstratos

### 1.1 Métodos e propriedades abstratos

**Métodos abstratos** são métodos **declarado na classe**, mas sua implementação é deixada para as subclasses.

Na declaração de um método abstrato em C# não se fornece nenhuma implementação real, não existe nenhum corpo de método; a declaração de método simplesmente acaba com um ponto-e-vírgula e não existe nenhuma chave ({ }) seguindo a assinatura. Exemplo:

```
public abstract void MyMethod();
```

A implementação é fornecida por um método de sobrecarga, que é um membro de uma classe não abstrata.

É um erro utilizar os modificadores estático ou virtual na declaração de um método abstrato.

Propriedades abstratas se comportam como métodos abstratos, exceto para as diferenças na sintaxe declaração e chamada.

É um erro usar o modificador abstract em uma propriedade estática.

## 1.2 Classes abstratas

**Classes abstratas** definem apenas parte da implementação, ou seja, definem métodos sem implementação (abstratos), os quais devem ser redefinidos em classes derivadas concretas.

Todos os métodos abstratos devem ser **implementados nas** subclasses **concretas**, ou seja, as subclasses devem definir a parte que está faltando.

**Classe abstrata** é um tipo de classe que somente **pode ser herdada e não instanciada**, de certa forma, pode-se dizer que este tipo de classe é uma **classe conceitual** que pode definir funcionalidades para que as suas subclasses (classes que herdam desta classe) possam implementa-las de forma não obrigatória, ou seja ao se definir um conjunto de métodos na classe abstrata não é obrigatório a implementação de todos os métodos em suas subclasses.

Classes abstratas **não podem ser instanciadas**, ou seja, nenhum objeto desta classe pode ser construído com a cláusula new.

Em uma classe abstrata os métodos declarados podem ser abstratos ou não, e suas implementações devem ser obrigatórias na subclasse ou não, quando se cria um método abstrato em uma classe abstrata sua implementação é obrigatória, caso não se implemente o mesmo o compilador gera um erro em tempo de compilação.

Classes abstratas permitem que se definam métodos sem implementação que devem ser redefinidos em classes derivadas.

O que define se uma classe é abstrata ou não, é a ocorrência de pelo menos um método abstrato.

Uma classe é considerada abstrata se tiver pelo menos um método abstrato, ou seja, classes abstratas normalmente possuem um ou mais métodos abstratos.

Se uma classe possui um método abstrato, ela deve ser necessariamente uma classe abstrata.

Uma classe abstrata não precisa possuir métodos abstratos. Mas toda classe com métodos abstratos deve ser declarada como uma classe abstrata.

As classes derivadas de classes abstratas herdam todos os métodos, incluindo os abstratos.

As classes derivadas de classes abstratas são abstratas até que implementem os métodos abstratos.

**Classes abstratas em C#** são declaradas com a **palavra-chave: abstract.**

Toda classe com **métodos abstratos ou virtuais em C#** deve ser declarada como uma classe abstrata.

**Exemplo: Classe abstrata** Forma em C#

```
public abstract class Forma    {
    private double area, double perimetro;
    private string cor;
```

```
    public string Cor{  get  {  return cor; } set  {  cor = value; }   }
    public double Area{  get  {return area;}  set   { area = value;} }
    public double Perimetro  {
          get {return _perimetro;}
          set  { perimetro = value; }
    }
    public abstract void CalcularPerimetro( ); // metodo abstrato
    public virtual void CalcularArea( ) {this.Area = 0;} // método virtual
}
```

**Exemplo: Classe concreta Quadrado** implementa Forma

```
public class Quadrado: Forma   {
    private double lado;
    public double Lado {  get  {return lado;}  set  {lado = value;} }
    public override void CalcularArea( )  {  this.Area = lado * lado;    }
    public override void CalcularPerimetro( )  {  this.Perimetro = 4 * lado;  }
}
```

**Exemplo: Classe concreta Circulo** implementa Forma

```
public class Circulo: Forma {
  private double raio, x , y;
  public Circulo (int x, int y, double raio) { this.x = x; this.y = y;  this.raio = raio;}
  public double Raio {
       get { return this.raio; }
       set { if ( value >= 0 )  { this.raio = value;  }

  }
  public double X {
       get { return this.x; }
       set {  this.x = value;  }
  }
 public double Y {
       get { return this.y; }
       set { this.y = value;  }
 }

public double Diametro( ) { return raio * 2;   }
public override void CalcularArea( )  { this.Area = Math.PI * Math.Pow( raio, 2 ); }
public override void CalcularPerimetro( ) { this.Perimetro = Math.PI * Diametro();  }

}
```

### 1.3 Métodos virtuais

**Métodos virtuais** podem ser reescritos na classe derivada.

Na **sobreposição (*overriding*),** os métodos com mesmo nome e assinatura idênticas.

**Método: Object.ToString**

É um método virtual que retorna um string que representa o objecto corrente, ou seja, converte um objeto em sua representação de cadeia de caracteres de modo adequado para exibição:

**public virtual string ToString( )**

A implementação padrão do método de ToString retorna o nome totalmente qualificado do tipo de Object.

É o principal método de formatação no .NET Framework.

**Exemplo1:**

```
public class Exemplo1 {
   public static void Main()   {
      Object obj = new Object();
      Console.WriteLine(obj.ToString());
   }
}
// O exemplo 1 imprime na tela:
//      System.Object
```

**Exemplo2:**

```
namespace Examples {
    public class Object1   {    }
}
public class Exemplo2 {
   public static void Main( )   {
      object obj1 = new Object1();
      Console.WriteLine(obj1.ToString());
   }
}
// O exemplo 2 imprime na tela :
//   Examples.Object1
```

**Exemplo3:**

```
public class Object2 {
   private object value;
   public Object2(object value)  {
      this.value = value;
   }
   public override string ToString() {
      return base.ToString() + ": " +  value.ToString();
   }
}

public class Example {
   public static void Main()    {
```

```
    Object2 obj2 = new Object2('a');
    Console.WriteLine(obj2.ToString());
  }
}
// O exemplo 3 imprime na tela :
//       Object2: a
```

## 1.4 Classes e membros de classes lacradas (*sealed*) em C#

A palavra-chave **sealed (lacrada)** permite evitar a herança de uma classe ou membros da classe que foram anteriormente marcados por virtual, ou seja, as classes lacradas impedem a derivação.

Uma classe **lacrada ou fecha**da não pode ser usada como classe base e não pode ser também uma classe abstrata.

Algumas otimizações de tempo de execução podem agilizar as chamadas aos membros de classe lacrada.

Uma **classe selada (*sealed*) em C#** é utilizada para restringir caracteristicas da herança do objeto, quando uma classe é definida como ***sealed***, esta classe não poderá ser herdada e a tentativa de herdar gera erro em tempo de compilação.

Após criar uma classe selada pode-se observar que o intelisense não mostra o nome da classe definida como sealed quando se tenta criar uma herança para novas classes.

Um **método em uma classe derivada** que está substituindo um membro virtual da classe base pode declarar esse **membro como sealed (lacrado**).

Isso nega o aspecto virtual de membro para qualquer classe derivada adicional.

As classes podem ser declaradas como lacradas inserindo a palavra-chave sealed antes da definição de classe.

**public sealed** class D {    // Class members here. }

**Exemplo 1:**

```
// Retirado do books online em C#.
sealed class ClasseSelada {
   public int x;
   public int y;
}

class MainClass {
   static void Main( )   {
      ClasseSelada sc = new ClasseSelada();
      sc.x = 110;
      sc.y = 150;
      Console.WriteLine("x = {0}, y = {1}", sc.x, sc.y);
```

```
   }
}
```

**Exemplo 2:**

```
class X    {
       protected virtual void F( ) { Console.WriteLine("X.F"); }
       protected virtual void F2( ) { Console.WriteLine("X.F2"); }
}

class Y : X    {
       sealed protected override void F( ) {
          Console.WriteLine("Y.F");
       }
       protected override void F2( ) {
           Console.WriteLine("Y.F2");
       }
       public void F3(){ F();}
       public void F4(){ F2();}
}

class Z : Y  {
       // Attempting to override F causes compiler error CS0239.
       protected override void F() { Console.WriteLine("C.F"); }
       // Overriding F2 is allowed.
       protected override void F2() { Console.WriteLine("Z.F2"); }
}
class MainClass {
   static void Main()    {
       Y sc = new Y();
       sc.F3();
       sc.F4();
   }
}
```

**Ex1.1** Escrever um **programa** codificados em **C#** para utilizar os conceitos de herança e classes abstratas.

1.1.1 Definir a classe abstrata Conta.

```
abstract class Conta {
          public double Saldo { get ; set ; }
}
```

1.1.2 Defina uma classe para modelar a contas de poupança

```
 class ContaPoupanca : Conta {
          public int DiaDoAniversario { get ; set ; }
 }
```

1.1.3

Altere a classe TestaConta para corrigir o erro de compilação.

```
class TestaConta {
```

```
        static void Main ( ) {
                Conta c = new ContaPoupanca ();
                c. Saldo = 1000;
                System . Console . WriteLine (c. Saldo );
        }
}
```

1.1.4 Defina ummétodo abstrato na classe Conta para gerar extratos detalhados.

```
abstract class Conta 2 {
        public double Saldo { get ; set ; }
        public abstract void ImprimeExtratoDetalhado ();
}
```

1.1.5 O que acontece com a classe ContaPoupanca?

1.1.6 Defina uma implementação do método **ImprimeExtratoDetalhad**o() na classe ContaPoupanca.

```
class ContaPoupanca : Conta {
   public int DiaDoAniversario { get ; set ; }
   public override void ImprimeExtratoDetalhado ( )  {
        System.Console.WriteLine (" EXTRATO DETALHADO DE CONTA POUPANÇA ");
        System . DateTime agora = System . DateTime . Now ;
        System . Console . WriteLine (" DATA : " + agora . ToString ("D"));
        System . Console . WriteLine (" SALDO : " + this . Saldo );
        System . Console . WriteLine (" ANIVERSÁRIO : " + this . DiaDoAniversario );
   }
}
```

1.1.**7 Altere a classe TestaConta para chamar o método ImprimeExtratoDetalhado().**

**Ex1.2 (exercício para entregar):** Escrever um **programa** codificados em **C#** para utilizar os conceitos de herança e classes abstratas para calcular o Imposto de Renda de uma coleção de contribuintes, que podem ser pessoas físicas ou pessoas jurídicas.

O cálculo do IR deve ser feito da seguinte maneira:

**Pessoa Jurídica**

O imposto deve corresponder a 10% da renda bruta da empresa.

**Pessoa Física**

O imposto deve ser calculado de acordo com a seguinte tabela:

**Renda Bruta Alíquota Parcela a Deduzir**

R$ 0,00 a R$ 1.400,00 0% R$ 0,00
R$ 1.400,01 a R$ 2.100,00 10% R$ 100,00
R$ 2.100,01 a R$ 2.800,00 15% R$ 270,00
R$ 2.800,01 a R$ 3.600,00 25% R$ 500,00
R$ 3.600,01 ou mais 30% R$ 700,00

As classes a seguir devem ser usadas conjuntamente com este enunciado. Elas contêm parte do código necessário à implementação deste exercício. Deve-se completá-las nos pontos indicados, de acordo com os objetivos do exercício.

```
public class PFisica : Contribuinte {
        protected String cpf;
        protected double salario;
        public PFisica(String n,string end, double sal,String c){
        // inicialização das varáveis de instância
```

```
        }
        public double calcImposto() {
        // Cálculo do imposto
        }
}
public class PJuridica : Contribuinte {
        protected String cnpj;
        protected double faturamento;
        public PJuridica(String n,string end, double f,String c){
                // inicialização das varáveis de instância
        }
        public double calcImposto(){
        // Cálculo do imposto
        }
}

public abstract class Contribuinte {
        protected String nome;
        protected String endereco;
        public static Contribuinte [] listaContr(){
                Contribuinte []lst = new Contribuinte[6];
                lst[0]=new PFisica("Joao Santos","Rua abc, 123",3000.00,"11111");
                lst[1]=new PJuridica("Lojas AA","Rua Hum, 111",150000.00,"10055");
                lst[2]=new PFisica("Maria Soares","Av. Xyz, 777",5000.00,"22222");
                lst[3]=new PJuridica("Supermercados B","Rua Dois, 987",2000000.00,"10066");
                lst[4]=new PFisica("Carla Maia","Av. Três, 333",1500.00,"33333");
                lst[5]=new PJuridica("Posto XX","Rua Cinco, 555",500000.00,"10077");
                return lst;
        }
        public String getNome(){
                return nome;
        }
        abstract public double calcImposto();

}
```

## 2. Interfaces

**Interface** é uma coleção de declarações de métodos, sem definição, que podem ser incorporados por classes.

É similar ao conceito de classe abstrata e **define as operações** que um objeto é **obrigado a implementar**.

**Nunca contém implementação**, ou seja, não pode definir campos.

Cada operação declarada deve especificar o nome da operação, os objetos que esta operação aceita como parâmetros e o tipo de valor retornado pela operação.

Este conjunto de informações sobre uma determinada operação tem o nome de **assinatura da operação**, e um **conjunto de assinaturas** de operações dá-se o nome de **interface**.

**Não permite construtores** pois um construtor tem as instruções usadas para inicializar campos.

Para usar uma interface deve-se **criar uma classe ou estrutura** e **herdar da interface**, com isso é obrigatório implementar todos os métodos da interface.

Deve-se usar uma **interface para simular a herança para estruturas**, porque realmente não podem herdar de outra estrutura ou classe.

Uma classe **herda apenas constantes** de uma interface.

Uma classe **não pode herdar implementações** de uma interface.

A hierarquia de interfaces é independente da hierarquia de classes.

Classes que implementam a mesma interface podem ou não estar relacionadas na hierarquia.

As interfaces são fundamentais em um sistema orientado a objetos, quando se diz que um objeto é a instância de uma classe, na verdade quer-se dizer, que este objeto implementa a interface definida pela classe, ou seja, uma interface define as operações que um objeto será obrigado a implementar.

Interfaces servem para definir protocolos de comportamento que possam ser implementados em qualquer lugar na hierarquia de classes e são úteis para:

- Capturar similaridades entre classes não relacionadas.
- Declarar métodos que uma ou mais classes devem inevitavelmente implementar.
- Revelar interfaces sem revelar os objetos que a implementam.
- Quando as alterações podem afetar sem querer as subclasses de execução
- Quando existem alterações que podem afetar o comportamento de forma indesejada.

Pode-se assumir que uma **interface é uma espécie de contrato** entre a classe e o mundo exterior.

Tal definição se dá pelo fato do compromisso assumido por uma classe quando ela implementa uma interface, fornecendo o comportamento determinado por ela.

Uma interface possui apenas métodos não implementados e que devem ser implementados pela classe que usar a interface.

A utilização de interface permite que uma classe estenda múltiplas interfaces para simular tal comportamento

Em C# e Java não é possível fazer herança múltipla.

Várias classes podem implementar de forma distinta uma mesma interface, permitindo uma forma alternativa de herança, pois **C# e Java permitem herança múltipla de interfaces**.

**Interfaces no C#:**

- não podem conter atributos
- somente pode ter métodos, propriedades e eventos
- Por default, todos os membros da interface são públicos
- Não permitem a utilização de um modificador

**Define-se** uma **interface em C#** usando a **palavra chave interface**. O nome da classe e o nome da interface são separados por dois pontos ": "

**Exemplo 2.1**: **Implementação Interface em C#**

```
interface IExemploInterface {
        void ExemploMetodo();
}
class Implementacaoclasse : IExemploInterface{
   void IExemploInterface.ExemploMetodo()  {  // método }

static void Main(){// Declar uma instancia de uma interface
   IExemploInterface obj = new Implementacaoclasse();
   obj.SampleMethod(); // chamar um método.
  }
}
```

**Exemplo 2.2**

```
public interface IVoar {
void LevantarVoo(aviao);
}

//Implementação desta interface
public class 14Bis : IVoar {
        public void LevantarVoo(aviao) {
                Console.Write("ao infinito e além...");
        }
}
```

**Exemplo2.3 : interface IForma em C#  public interface IForma** {

```
// classes que implement IForma precisam implementar estes métodos e propriedades
        double Area();
        double Perimetro();
        string Name { get; }
}

// Classe Ponto que implementa IForma
public class Ponto: IForma {
        private int x, y; // coordrnadas
        public Ponto( int xValue, int yValue )  {// constructor
                X = xValue;  Y = yValue;
        }
        public int X   {           // propriedade X
                get  {  return x;  } set  {  x = value; }
        }
        public int Y  {// propriedade y
                        get { return y; }          set { y = value; }
        }

        public virtual double Area() { return 0; } // implementa metodo Area da interface IForma

        public virtual double Perimetro() {  return 0; } // implementa metodo Perimetro de IForma
```

```
        public virtual string Name {
                get { return "Ponto"; } // implementa propriedade  Name  de IForma
        }
} // end class Ponto


// Classe concreta Circulo estende Ponto
public class Circulo: Ponto {
        private double raio; // raio do circulo
        public Circulo(int xc, int yc, double rc): base( xc, yc ) {
                Raio = rc;
         }
        public double Raio {
                get { return this.raio; }
                set {  if ( value >= 0 )   this.raio = value;       }
        }
        public double Diametro()  {return Raio * 2; }

        public override double Area() {
                return (Math.PI * Math.Pow( raio, 2 ));
        }
        public override double Perimetro() {
                return (Math.PI * Diametro());
        }
        public override string ToString(){
                return "Centro = [" + X + ", " + Y + "]" + "; Raio = " + raio;
        }
        public override string Name {  get  {  return "Circulo"; }
}
```

## 2.1 Interfaces C# do namespace System.Collections

O **namespace System.Collections** contém interfaces e classes que definem várias coleções de objetos, como listas, filas, tabelas hash e dicionários.

**Exemplos** de algumas interfaces:

**Icollection:** interface base para classes na System.Collections e define tamanho, enumeradores e métodos de sincronização para todas as coleções não genéricas.

**Icomparer:** expõe um método que compara dois objetos.

**Ienumerable:** expõe um enumerador, que oferece suporte a uma iteração simples sobre uma coleção não genérica.

**Ilist:** coleção de objetos não-genéricos que podem ser acessados individualmente pelo índice.

**Ex2.1 (exercício para entregar):** Criar uma interface controle remoto com os metodos ligar e desligar. A partir desta interface criar as classes **Televisor** e **DVD**. A classe **Televisor** deve ter métodos para ligar e desligar, aumentar ou diminuir o volume (com mínimo de 0 e máximo de 100)

e subir ou baixar o canal (entre 1 e 83). Considere acessos públicos e privados, bem como métodos *getters* e *setters.*

A partir destas classes escrever **programas** codificados em **C# e** para testar as funcionalidades das classes obtidas.

**Ex2.2 (exercício para entregar):**
Crie a seguinte hierarquia de classes em C#:

- Uma interface para representar qualquer forma geométrica, definindo métodos para cálculo do perímetro e cálculo da área da forma;
- Classes para representar retângulos e quadrados.
- A primeira deve receber o tamanho da base e da altura no construtor, enquanto a segunda deve receber apenas o tamanho do lado;
- Uma classe para representar um círculo. Seu construtor deve receber o tamanho do raio.

No programa principal em C#, pergunte ao usuário quantas formas ele deseja criar.
Em seguida, para cada forma, pergunte se deseja criar um quadrado, um retângulo ou um círculo, solicitando os dados necessários para criar a forma.
Todas as formas criadas devem ser armazenadas em um vetor.
Finalmente, imprima:
(a) Os dados (lados ou raio);
(b) Os perímetros;
(c) As áreas de todas as formas.

## 3. Polimorfismo (*polymorphism*)

**Polimorfismo (*polymorphism*)** é uma característica de linguagens orientadas a objetos que permite que diferentes objetos respondam a mesma mensagem, cada um a sua maneira.

Polimorfismo é o efeito produzido por uma função (ou método) capaz de operar para uma variedade de tipos de dados (ou classes de objetos).

Este princípio permite que classes filhas tenham métodos iguais, mas comportamentos diferentes, ou seja, métodos iguais com a mesma assinatura e comportamentos diferentes implicam em ações diferentes.

Nas linguagens orientadas a objetos o efeito de polimorfismo está intimamente associado aos conceitos de tipo abstrato de dados e de herança.

No **polimorfismo universal** ou polimorfismo verdadeiro (*true polymorphism*) ocorre quando uma função trabalha uniformemente para uma gama de tipos; tipos esses normalmente com uma estrutura uniforme.

Um mesmo método (ou função) pode ser executado para um conjunto possivelmente ilimitado de tipos de dados diferentes.

### 3.1 Namespace System.Collections C#

Contém interfaces e **classes** que definem várias coleções de objetos, como listas, filas, matrizes de bits, tabelas de hash e dicionários. Exemplos de algumas classes:

**ArrayList**: implementa a interface **IList** usando um vetor cujo tamanho é aumentado dinamicamente quando necessário.

**Hashtable:** Representa uma coleção de chave/valor pares que são organizados com base na código hash da chave.

**Queue:** representa uma coleção *first-in, first-out (primeiro entrar, primeiro a sair)* dos objetos.

**Stack:** Representa uma simples coleção de objetos não genéricos last-in-first-out (LIFO).

**Alguns métodos da Classe ArrayList**

- **Add:** adiciona itens ao final do ArrayList.
- **Contains**: determina se um elemento está no ArrayList. testa a presença de um item na lista.
- **Insert:** insere um elemento em ArrayList no índice especificado.
- **Remove:** remove a primeira ocorrência de um objeto específico do ArrayList.
- **Clear:** remove todos os elementos de ArrayList.

**Exemplo com ArrayList:**

```
using System;
using System.Collections;
public class SamplesArrayList  {
  public static void Main()  {
    ArrayList myAL = new ArrayList(); // Criar ArrayList
    myAL.Add("Hello");
    myAL.Add("World");
    myAL.Add("!");
    Console.WriteLine( "myAL" ); // Mostrar as propridades e valores do ArrayList.
    Console.WriteLine( "    Count:    {0}", myAL.Count );
    Console.WriteLine( "    Capacity: {0}", myAL.Capacity );
    Console.Write( "    Values:" );
    PrintValues( myAL );
  }
  public static void PrintValues(IEnumerable myList )  {
        foreach ( Object obj in myList )
              Console.Write( "   {0}", obj );
        Console.WriteLine();
  }
}
```

**Imprime na Tela:**

```
myAL
Count:    3
Capacity: 4
Values:   Hello   World   !
```

### 3.2 Genéricos (*generics*) do C#

Os Generics foram adicionados à versão 2.0 da linguagem C# e o Common Language Runtime (CLR).

Generics introduzem no .NET Framework o conceito de parâmetros de tipos, que tornam possíveis a estruturação de classes e métodos que adiam a especificação de um ou mais tipos até que a classe ou método seja declarada e instanciada.

Usando um parâmetro de tipo genérico T pode-se escrever uma única classe que outro código do cliente poderá usar sem aumentar o custo ou risco de conversões (cast) em tempo de execução (runtime).

#### 3.2.1 Namespace System.Collections.Generic C#

Contém interfaces e classes que definem **coleções genéricas**, que fornecem melhor segurança de tipo e desempenho do que coleções fortemente não-genéricas. Exemplos de algumas classes:

**IList<T>:** representa uma lista de objetos que podem ser acessados pelo índice. Fornece métodos para pesquisar, ordenar e manipular listas.

**HashSet<T>:** representa um conjunto de valores.

**Queue <T>:** representa uma coleção *first-in, first-out (primeiro entrar, primeiro a sair)* de objetos genéricos.

**Stack <T>:** representa uma simples coleção de objetos genéricos last-in-first-out (LIFO).

#### 3.2.2 Classe List<T>

A **classe List<T>** é o equivalente genérico da classe de ArrayList .

Implementa a interface genérica de IList<T> usando uma matriz cujo tamanho é gerado dinamicamente conforme necessário.

Namespace System.Collections.Generic;

public class List<T>: IList<T>, ICollection<T>, IReadOnlyList<T>, IReadOnlyCollection<T>, IEnumerable<T>

- O método de Add adiciona itens a List<T> ,
- Contains testa a presença de um item na lista,
- Insert insere um novo item no meio da lista,
- Remove remove a primeira instância item duplicado, anteriormente adicionado
- Clear remove todos os itens na lista.

**Genéricos (*generics*): exemplo**

```
public class GenericList<T> {Declare the generic class
   void Add(T input) { }
}
class TestGenericList {
```

```
    private class ExampleClass { }
    static void Main()   {
      GenericList<int> list1 = new GenericList<int>(); // list of type int.
      // list of type string.
      GenericList<string> list2 = new GenericList<string>();
      // list of type ExampleClass.
       GenericList<ExampleClass> list3 = new GenericList
                                              <ExampleClass>();
    }
}
```

### 3.3 Polimorfismo universal Paramétrico

A capacidade de tratar objetos criados a partir das classes específicas como objetos de uma classe genérica é chamada de **polimorfismo**.

Utiliza tipo abstrato de dados ou métodos cujos códigos são *genéricos*, de maneira a permitir que valores sejam manipulados de forma similar independentemente do seu tipo.

A *genericidade* é atingida por:

- Uso de classes ou tipos básicos como parámetro
- Mecanismo de implementação que substitua implícita ou explicitamente o tipo parametrizado quando necessário

Ao definir um elemento (que pode ser uma classe, um método ou alguma outra estrutura da linguagem), a definição do tipo é incompleta e precisa parametrizar este tipo.

**Polimorfismo universal paramétrico: exemplo 1**

```
using System;
using System.Collections.Generic;
....
static void Main(string[ ] args)   {
  List<Object> LISTA = new List<Object>();
   int T;  string N;
   Console.Write("Quantos nomes a entrar? ");
   T = int.Parse(Console.ReadLine());
   for (int i = 0; i < T; i++) {// Entrada dos nomes
         Console.Write("Entre o {0,3}o. nome: ", i + 1);
         N = Console.ReadLine();
         LISTA.Add(N);
   }
```

```
    for (int i = 0; i < LISTA.Count; i++) { // Apresentacao dos nomes
        N = LISTA[i].ToString( );
        Console.WriteLine("{0,3}o. nome = {1}", i + 1, N);
    }
}
```

**Polimorfismo universal paramétrico: exemplo 2**

Com o intuito inicial de reutilizar código, podemos modelar os diversos tipos de funcionários do banco utilizando o conceito de herança.

```
class Funcionario {
        public int Codigo { get ; set ; }
}

class Gerente : Funcionario {  // TODO Gerente É UM Funcionario

        public string Usuario { get ; set ; }
        public string Senha { get ; set ; }
}

class Telefonista : Funcionario {
        public int Ramal { get ; set ; }
 }
```

Além de gerar reaproveitamento de código, a utilização de herança permite que objetos criados a partir das classes específicas sejam tratados como objetos da classe genérica.

Em outras palavras, a herança entre as classes que modelam os funcionários permite que objetos criados a partir das classes Gerente ou Telefonista sejam tratados como objetos da classe Funcionario.

No código da classe Gerente utiliza-se o símbolo "**:**¨ para indicar que a classe Gerente é uma subclasse de Funcionario. Esse símbolo pode ser interpretado como a expressão: **É UM** ou **É UMA**.

Como está explícito no código que todo gerente éumfuncionário então podemos criar um objeto da classe Gerente e tratá-lo como um objeto da classe Funcionario também.

```
// Criar um objeto da classe Gerente
Gerente g = new Gerente ();
// Tratar um gerente como um objeto da classe Funcionario
Funcionario f = g;
```

Em alguns lugares do sistema do banco será mais vantajoso tratar um objeto da classe Gerente como um objeto da classe Funcionario.

O registro da entrada ou saída não depende do cargo do funcionário. Não faz sentido criar um método que registre a entrada para cada tipo de funcionário, pois eles serão sempre idênticos.

Analogamente, não faz sentido criar um método que registre a saída para cada tipo de funcionário.

Dado que pode-se tratar os objetos das classes derivadas de Funcionario como sendo objetos dessa classe, pode-se implementar um método que seja capaz de registrar a entrada de qualquer funcionário independentemente do cargo. Analogamente, pode-se fazer o mesmo para o procedimento de saída.

```
using System ;

 class ControleDePonto {
        public void RegistraEntrada ( Funcionario f) {
                DateTime agora = DateTime . Now ;
                string horario = String . Format (" {0: d/M/ yyyy HH:mm:ss}", agora );
                System . Console . WriteLine (" ENTRADA : " + f. Codigo );
                System . Console . WriteLine (" DATA : " + horario );
        }

        public void RegistraSaida ( Funcionario f) {
                DateTime agora = DateTime . Now ;
                string horario = String . Format (" {0: d/M/ yyyy HH:mm:ss}", agora );
                System . Console . WriteLine (" SAÍDA : " + f. Codigo );
                System . Console . WriteLine (" DATA : " + horario );
        }

 }
```

Os métodos RegistraEntrada() e RegistraSaida() recebem referências de objetos da classe Funcionario como parâmetro. Consequentemente, podem receber referências de objetos de qualquer classe que deriva direta ou indiretamente da classe Funcionario.

Aplicando a ideia do polimorfismo no controle de ponto, facilita-se a manutenção da classe ControleDePonto. Qualquer alteração no procedimento de entrada ou saída implica em alterações em métodos únicos.

Além disso, novos tipos de funcionários podem ser definidos sem a necessidade de qualquer alteração na classe ControleDePonto. Analogamente, se algum cargo deixar de existir, nada precisará ser modificado na classe ControleDePonto.

**Ex3.1 (exercício para entregar):**
3.1.1 Defina uma classe para modelar de forma genérica os funcionários do banco.
3.1.2 Implemente duas classes específicas para modelar dois tipos particulares de funcionários do banco: os gerentes e as telefonistas.
3.1.3 Implemente o controle de ponto dos funcionários. Crie uma classe com dois métodos: o primeiro para registrar a entrada dos funcionários e o segundo para registrar a saída.
3.1.4 **Fazer um programa de teste em C# para testar** a lógica do **controle de ponto**, registrando a entrada e a saída de um **gerente** e de uma **telefonista**.

**Ex3.2 (exercício para entregar):**
3.2.1 A partir da classe genérica para modelar as contas do banco.

```
class Conta {
        public double Saldo { set ; get ; }
}
```

3.2.2 Definir duas classes específicas para dois tipos de contas do banco: poupança e corrente.

```
class ContaPoupanca : Conta  {
        public int DiaDoAniversario { get ; set ; }
 }
class ContaCorrente : Conta {
        public double Limite { get ; set ; }
}
```

3.2.3 Definir uma classe para especificar um gerador de extratos.

```
using System ;
class GeradorDeExtrato {
        public void ImprimeExtratoBasico ( Conta c) {
                DateTime agora = DateTime . Now ;
                string horario = String . Format (" {0: d/M/ yyyy HH:mm:ss}", agora );
                System . Console . WriteLine (" DATA : " + horario );
                System . Console . WriteLine (" SALDO : " + c. Saldo );
        }
}
```

3.2.4 **Fazer um programa de teste em C#** para o **gerador de extratos**.

**Ex3.3 (exercício para entregar):**

3.3.1 Definir uma interface para padronizar as assinaturas dos métodos das contas do banco.

```
interface IConta {
        void Deposita ( double valor );
        void Saca ( double valor );
        double Saldo { get ; set ; }
}
```

3.3.2 Crie as classes a seguir  para modelar tipos diferentes de conta.

```
class ContaCorrente : IConta {
        public double Saldo { get ; set ; }
        private double taxaPorOperacao = 0.45;
        public void Deposita ( double valor ){
                this . Saldo += valor - this . taxaPorOperacao ;
        }
        public void Saca ( double valor ){
                this . Saldo -= valor + this . taxaPorOperacao ;
        }
}

class ContaPoupanca : IConta {
        public double Saldo { get ; set ; }
        public void Deposita ( double valor ){
                this . Saldo += valor ;
        }
        public void Saca ( double valor ) {
                this . Saldo -= valor ;
        }
```

```
}
```

3.3.3 Crie um gerador de extratos com um método que pode trabalhar com todos os tipos de conta.

```
class GeradorDeExtrato {
        public void GeraExtrato ( IConta c)   {
                System . Console . WriteLine (" EXTRATO ");
        System . Console . WriteLine (" SALDO : " + c. Saldo );
        }
}
```

3.3.4 Fazer um **programa codificado em C#** que implemente estas **classes** e apresente na tela um menu com as seguintes opções:
1. Criar uma conta nova.
2. Excluir uma conta existente
3. Depositar em uma conta
4. Sacar de em uma conta
5. Imprimir o extrato de uma conta
6. Imprimir uma relação das contas existentes informando o número da conta e o nome do titular da conta
7. Sair do programa

O programa deve obter a opção do usuário, chamar o método correspondente, apresentar o resultado e sempre voltar ao menu inicial, exceto quando for selecionada a opção 7 (Sair do programa).

**Ex3.4 (exercício para entregar):** Utilize as classes *Telefone* e *TelefoneEletronico* do exerccío 5.1 (do LAB 1). A partir destas classes escrever um **programa** codificado em **C#** para ilustrar um método polimórfico. Faça a classe derivada sobrescrever (***override***) o método *Ring( )* para exibir uma mensagem diferente.

**Ex3.5 (exercício para entregar):**
Escrever um **programa** codificado em **C#** Implemente uma aplicação que declara uma variável polimórfica do tipo OperacaoMatematica.
A partir de dados fornecidos pelo usuário, a aplicação deve realizar uma operação matemática
E imprimir o seu resultado.

Ofereça ao usuário um menu para a escolha entre as operações matemáticas disponíveis.
OBS: Não defina a e b como atributos

Implemente um construtor padrão para cada uma das classes.