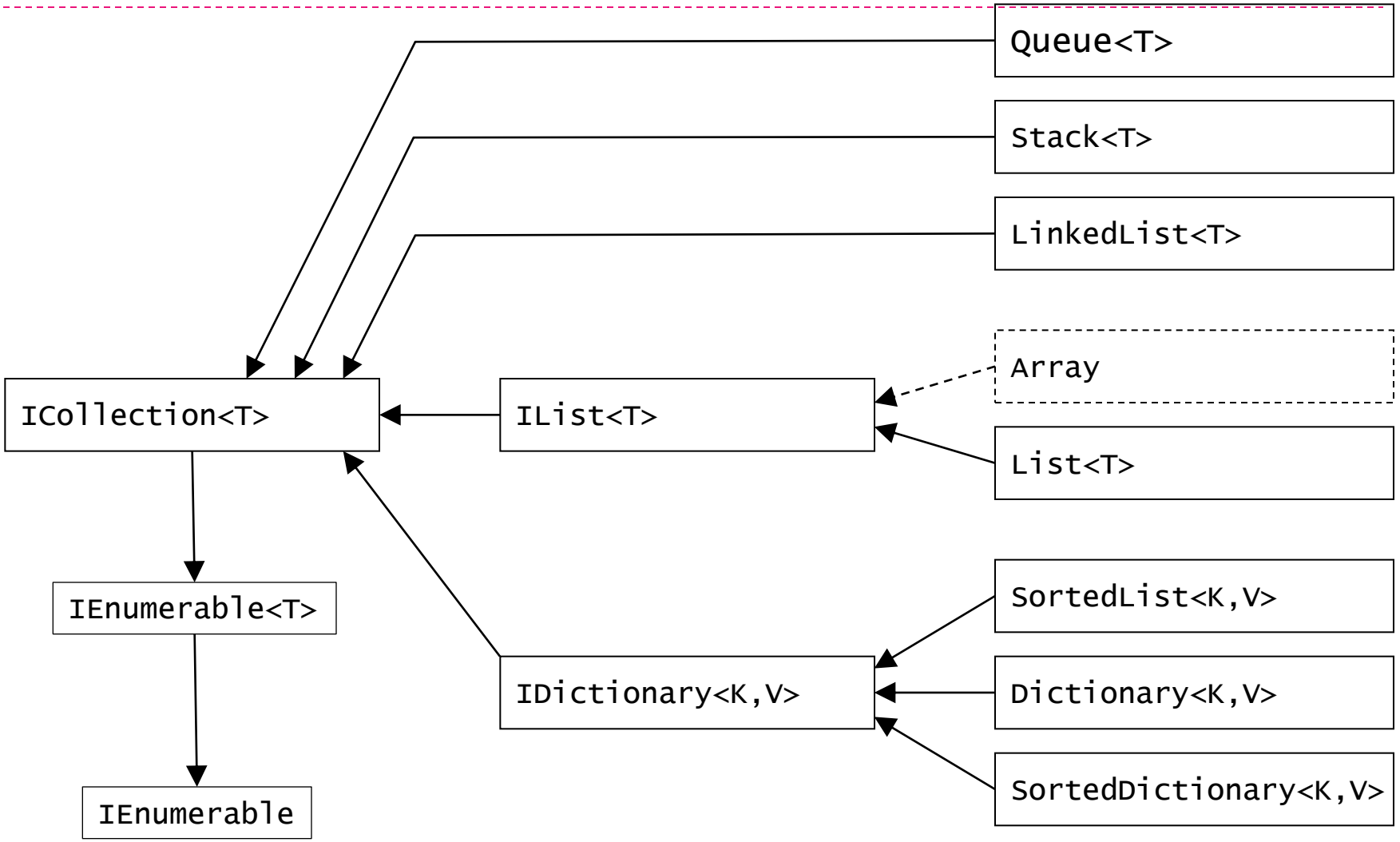
Ambientes Virtuais de Execução

Genericos (Continuação)

System.Collections.Generic (interfaces e implementações)



Retrocompatibilidade

- Versões genéricas de `IEnumerable` e de `IEnumerator` estendem as versões anteriores (não genéricas)

```
public interface IEnumerable<T> : IEnumerable
public interface IEnumerator<T> : IEnumerator, IDisposable
```

- Colecções genéricas implementam interfaces genéricas e não-genéricas

```
public class List<T> :
    IList<T>, ICollection<T>, IEnumerable<T>,
    IList, ICollection, IEnumerable
```

classe Array

```
public abstract class Array : ICloneable, IList, ICollection,
IEnumerable {
   public static void Sort<T>(T[] array);
  public static void Sort<T>(T[] array, IComparer<T> comparer);
  public static Int32 BinarySearch<T>(T[] array, T value);
  public static Int32 BinarySearch<T>(T[] array, T value,
        IComparer<T> comparer);
  ...
}
```

Métodos genéricos - Exemplo

```
private static void Swap<T>(ref T o1, ref T o2) {
        T temp = o1;
        o1 = o2;
        o2 = temp;
}
```

```
private static void CallingSwap() {
  Int32 n1 = 1, n2 = 2;
  Console.WriteLine("n1={0}, n2={1}", n1, n2);
  Swap<Int32>(ref n1, ref n2);
  Console.WriteLine("n1={0}, n2={1}", n1, n2);
  String s1 = "Aidan", s2 = "Kristin";
  Console.WriteLine("s1={0}, s2={1}", s1, s2);
  Swap<String>(ref s1, ref s2);
  Console.WriteLine("s1={0}, s2={1}", s1, s2);
}
```

Métodos genéricos - Exemplo

```
internal sealed class GenericType<T> {
  private T m_value;
  public GenericType(T value) { m_value = value; }
     public TOutput Converter<TOutput>() {
     TOutput result = (TOutput) Convert.ChangeType(m_value, typeof(TOutput));
     return result;
  }
}
```

```
public class Program{
public static void Main(){
  GenericType<int> a;
  a=new GenericType<int>(65);
  String p=a.Converter<String>();
  Console.WriteLine(p);
}
}
```

```
//…
public struct Ponto{
 public int x, y;
 public Ponto(int a, int b){ x=a;y=b;}}
 public class Program{
    public static void Main(){
      GenericType<int> a=new GenericType<int>(65);
        Ponto p=a.Converter<Ponto>();
       Console.WriteLine(p);
}}
```

Existem algum problema?

Tipos Genéricos e herança

```
internal sealed class Node<T> {
 public T m_data;
 public Node<T> m_next;
 public Node(T data) :this(data, null) {
 }
 public Node(T data, Node<T> next) {
   m_data = data; m_next = next;
 }
public override String ToString() {
  return m_data.ToString() +
    ((m_next != null) ?
      m_next.ToString() : String.Empty);
}
}
```

```
internal class Node {
 protected Node m_next;
 public Node(Node next) {m_next = next;}
}

internal sealed class TypedNode<T> : Node {
 public T m_data;
 public TypedNode(T data):this(data, null)
 {   }
public TypedNode(T data, Node next) :
                                  base(next)
   { m_data = data;}
 public override String ToString() {
  return m_data.ToString() +
        ((m_next != null) ? m_next.ToString()
             : String.Empty);
}}
```

Diferenças?

Tipos abertos e fechados

- A definição de um tipo genérico corresponde à criação de um novo tipo, mas do qual não se podem criar instâncias (tipo aberto)

- Para cada parametrização (completa) de um tipo aberto a máquina virtual (o *compilador JIT*) cria um correspondente tipo fechado do qual se podem criar instâncias

Exemplo

```
using System; using System.Collections.Generic;
internal sealed class DictionaryStringKey<TValue> : Dictionary<String, TValue> {   }

public static class Program {
  public static void Main() {
     Type t = typeof(Dictionary<,>); // Tipo aberto
     Object o = CreateInstance(t); // Falha
     t = typeof(DictionaryStringKey<>); //Tipo Aberto
     o = CreateInstance(t); //Falha
     t = typeof(DictionaryStringKey<Guid>); // Tipo Fechado
     o = CreateInstance(t); // Tem Sucesso
  }
 private static Object CreateInstance(Type t) {
 Object o = null;
 try {
  o=Activator.CreateInstance(t); Console.WriteLine("Created instance of {0}", t.ToString());
   } catch (ArgumentException e) { Console.WriteLine(e.Message); }
   return o;
 } }
```

Instruções IL e prefixos de instruções modificados e acrescentadas devido aos genéricos (excerto)

- initobj
- newobj
- constrained.
- box
- unbox.any

Exemplo 1

- Qual é o problema deste método?

```
private static T Min<T>(T o1, T o2) {
   if (o1.CompareTo(o2) < 0) return o1;
   return o2;
}
```

Exemplo 1 - continuação

- Solução – restringir o tipo T:

```
public static T Min<T>(T o1, T o2) where T : IComparable<T>
{
  if (o1.CompareTo(o2) < 0) return o1;
  return o2;
}
```

Exemplo 2

- CLR não permite sobrecarga baseada no nome dos parametros de tipo ou restrições

```
internal sealed class AType {}
internal sealed class AType<T> {}
internal sealed class AType<T1, T2>
{}

internal sealed class AType<T> where
T : IComparable<T> {}  //ERRO

internal sealed class AType<T3, T4>
{} //ERRO
```

```
internal sealed class AnotherType
{
 private static void M() {}
 private static void M<T>() {}
 private static void M<T1, T2>()
{}

private static void M<T>() where T
: IComparable<T> {} //ERRO

private static void M<T3, T4>() {}
//ERRO
}
```

Exemplo 2 - continuação

```
internal class Base {
  public virtual void M<T1, T2>() where T1 : struct where T2 :
class {  }
}

internal sealed class Derived : Base {

public override void M<T3, T4>() //PODE SER
     where T3 : EventArgs // ERRO
     where T4 : class // ERRO
{ }
```

Constraints (Restrições)

- Nos tipos genéricos do CTS:
  - A compilação é realizada sem o conhecimento do tipo parâmetro
  - Para validar a utilização é necessário *algum* conhecimento sobre o tipo
- Por omissão, os tipos parâmetro só podem ser usados através da interface de **object**, já que é a única que é garantidamente implementada.
- Podem ser aplicadas restrições (*constraints*) aos tipos-parâmetro:
  - classe base
  - interfaces implementadas
  - existência de construtor sem parâmetros (`new()`)
  - tipo-referência (`class`) ou tipo-valor (`struct`)

Constraints (Restrições) II

```
class Utils {
    public static T max<T>(params T[] vals) where T : IComparable<T> {
      if (vals.Length == 0)
          throw new System.Exception("Invalid Argument List");
      T r = vals[0];
      for(int i=1; i < vals.Length; ++i) {
         if (r.CompareTo(vals[i]) < 0) r = vals[i];
      }
      return r;
    }
}
```

Constraints	Descrição
where T: <className>	**T** tem de derivar de <className>
where T:<interfaceName>	**T** tem de implementar <interfaceName>
where T : class	**T** tem de ser um tipo referência
where T: struct	**T** tem de ser um tipo valor
where T : new()	**T** tem de ter construtor sem parâmetros

Conversão de um tipo genérico

- Converter uma variável do tipo genérico para outro tipo é ilegal a não ser que seja um tipo compatível com a uma restrição

```
private static void CastingAGenericTypeVariable1<T>(T obj) {
  Int32 x = (Int32) obj; // ERRO
  String s = (String) obj; // ERRO
 }

private static void CastingAGenericTypeVariable2<T>(T obj) {
  Int32 x = (Int32) (Object) obj; // SEM ERRO
  String s = (String) (Object) obj; // SEM ERRO
}

private static void CastingAGenericTypeVariable3<T>(T obj) {
  String s = obj as String; // SEM ERRO
}
```

Outras questões de verificabilidade

```
private static void SettingAGenericTypeVariableToNull<T>() {
  T temp = null;// ERRO
 }

private static void SettingAGenericTypeVariableToDefaultValue<T>()
{
 T temp = default(T); // OK
}
```

```
private static void ComparingAGenericTypeVariableWithNull<T>(T obj)
{
  if (obj == null) { /* Nunca executa para um tipo valor*/ }
}
```

Anatomia de um genérico

```
// Classe genérica com dois tipos-parâmetro: U e V
public class Generic<U,V>
   // O tipo U deve ter SomeBaseClass como classe base e implementar as
   // interfaces (também genéricas) ISomeInterface e IOneMoreInterface<V>
   where U : SomeBaseClass, ISomeInterface, IOneMoreInterface<V>
   // O tipo V deve ser um tipo-referência e implementar a interface
   // IAnotherInterface<U> e a interface genérica IYetAnotherInterface<V>
   where V : class, IAnotherInterface<U>, IYetAnotherInterface<V>, new()
{
   // Construtores não podem ser genéricos
   static Generic() { /* ... */ }
   public Generic() { /* ... */ }
   // Método não genérico: pode usar tipos-parâmetro da classe
   public void m1(U u,V v) { /* ... */ }

   // Método genérico com um tipo-parâmetro: X
   // Pode usar os seus tipos-parâmetro para além dos da classe
   public void m2<X>(X x, U u) { /* ... */ }

   // Classe interna com três tipos-parâmetro: U,V e W
   public class Nested<W> { public void m<Z>(V v,W w, Z z) { /* ... */ } }
}
```

Campos estáticos

- A definição de uma classe genérica representa um conjunto de tipos
  - cada um desses tipos é uma instância da classe genérica
  - `Stack<T> : { Stack<int>, Stack<string>, ... }`
- Cada instância de uma classe genérica tem um conjunto próprio de campos estáticos
- O construtor estático é chamado para cada instanciação da classe genérica

```
public static class Singleton<T> where T : new() {
   static Singleton() {}
   static public readonly T Instance = new T();
}

public static partial class Examples {
   public static void SingletonExample() {
      Singleton<StringBuilder>.Instance.Append("a");
      Singleton<StringBuilder>.Instance.Append("b");
      Console.WriteLine(Singleton<StringBuilder>.Instance);
   }
}
```

Exemplo

```
public static class Utils {
   class RangeComparer<T> where T : IComparable<T> {
      private T min, max;
      public RangeComparer(T mn, T mx) { min = mn; max = mx; }
      public bool IsInRange(T t) {
         return t.CompareTo(min) >= 0 && t.CompareTo(max) <= 0;
      }
   }
   public static T[] InRange<T>(T[] ts, T min, T max)
         where T : IComparable<T> {
      return Array.FindAll(ts, new RangeComparer<T>(min, max).IsInRange);
   }
}
public static class Examples {
   public static void Main() {
      Array.ForEach(
         Utils.InRange(new int[] { 10, 21, 32, 43 }, 20, 40),
         Console.WriteLine
      );
   }
}
```

Invariância de tipos genéricos

- *Invariant generic typing*

  `List<string>` não é um subtipo de `List<object>`

```
public static class Examples {
    public static void AddObjectToList(List<object> list) {
       list.Add(new object()); // lista de strings conteria um object
    }
    public static void Main() {
       List<string> list = new List<string>();
       list.Add("A"); list.Add("B");
       AddObjectToList(list); // não compila: lista poderia ser modificada
    }
}
```

- Alternativa
  - definição de método genérico

    void AddObjectToList<T>(List<T> list) [where T : SomeClass]$_{opt}$

Co-variância e Contra-invariância nos Genéricos

- Um parâmetro do tipo genérico por ser:
- Invariante
  - Significa que não pode ser mudado.
- Contra-variante
  - Significa que pode mudar de uma classe para uma classe derivada.
  - Isso é indicado em C# pela palavra in.
  - Apenas pode aparecer como input, tal como os argumentos de métodos.
- Co-variante
  - Significa que pode mudar de uma classe para uma das suas classes base
  - Isso é indicado em C# pela palavra out.
  - Apenas pode aparecer como output, tal como o tipo de retorno dos métodos

Co-variância nos Genéricos

```
using System;
using System.Collections.Generic;
class Base {
  public static void PrintBases(IEnumerable<Base> bases) {
    foreach(Base b in bases) { Console.WriteLine(b); }
  }
}
 class Derived : Base {
public static void Main() {
  List<Derived> dlist = new List<Derived>();
   Derived.PrintBases(dlist);
   IEnumerable<Base> bIEnum = dlist; }
}
```

Contra-variância nos Genéricos

```
using System; using System.Collections.Generic;
abstract class Shape { public virtual double Area { get { return 0; }} }
 class Circle : Shape {
  private double r;
  public Circle(double radius) { r = radius; }
  public double Radius { get { return r; }}
  public override double Area { get { return Math.PI * r * r; }}
}
class ShapeAreaComparer : System.Collections.Generic.IComparer<Shape> {
 int IComparer<Shape>.Compare(Shape a, Shape b) {
  if (a==null) return b==null ?0:-1; return b==null ? 1 : a.Area.CompareTo(b.Area);}
}
 class Program {
  static void Main() {
   SortedSet<Circle> circles= new SortedSet<Circle>(new ShapeAreaComparer())
   circles.add(new Circle(7.2)), circles.add(new Circle(100));
  foreach (Circle c in circles) { Console.WriteLine(c == null ? "null" :
                                          "Circle with area " + c.Area); } } }
```