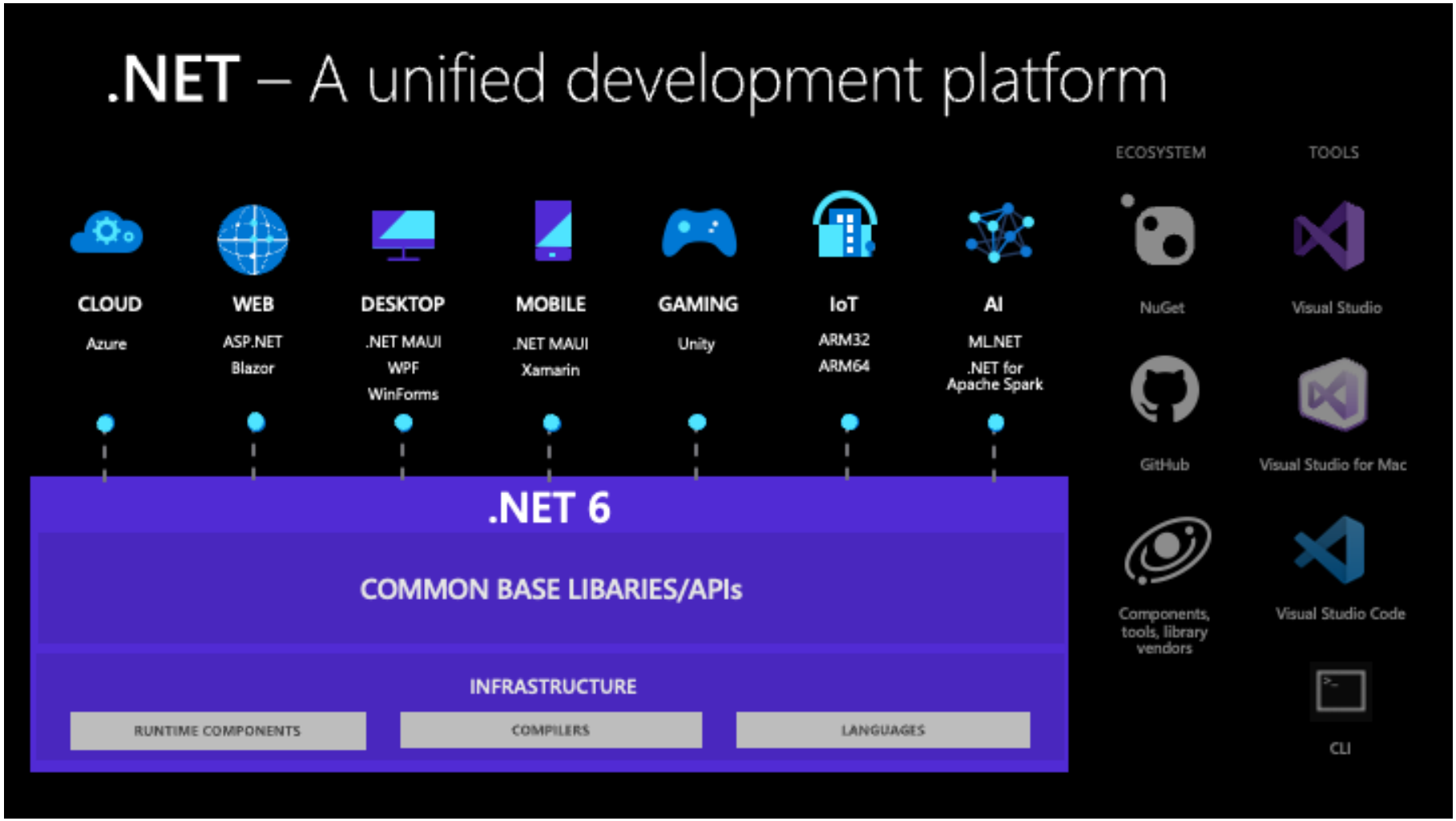
Dell IT Academy

Desenvolvimento de Sistemas

O que é o .NET?

https://dotnet.microsoft.com/

O que é o .NET?


.NET – A unified development platform
CLOUD
Azure
WEB
ASP.NET
Blazor
DESKTOP
.NET MAUI
WPF
WinForms
MOBILE
.NET MAUI
Xamarin
GAMING
Unity
IoT
ARM32
ARM64
AI
ML.NET
.NET for Apache Spark
ECOSYSTEM
TOOLS
NuGet
Visual Studio
GitHub
Visual Studio for Mac
Components, tools, library vendors
Visual Studio Code
CLI
.NET 6
COMMON BASE LIBARIES/APIs
INFRASTRUCTURE
RUNTIME COMPONENTS
COMPILERS
LANGUAGES

Qual versão do .NET?

- .NET 6
  - Versão multiplataforma
    - Windows, Linux, Mac OS
  - https://docs.microsoft.com/dotnet

Complementos ao .NET

- Utilizam a distribuição via *NuGet Package Manager*
  - https://www.nuget.org/
  - Gerenciamento de pacote via o próprio Visual Studio ou CLI

Qual versão do C#?

- C# 10
  - Não iremos explorar todas as características da sintaxe
  - https://dotnet.microsoft.com/en-us/languages/csharp

INTRODUÇÃO AO C#

Introdução ao C#

- É uma linguagem de programação Orientada a Objetos e Orientada a Componentes criada pela Microsoft para o .NET
- https://docs.microsoft.com/en-us/dotnet/csharp/

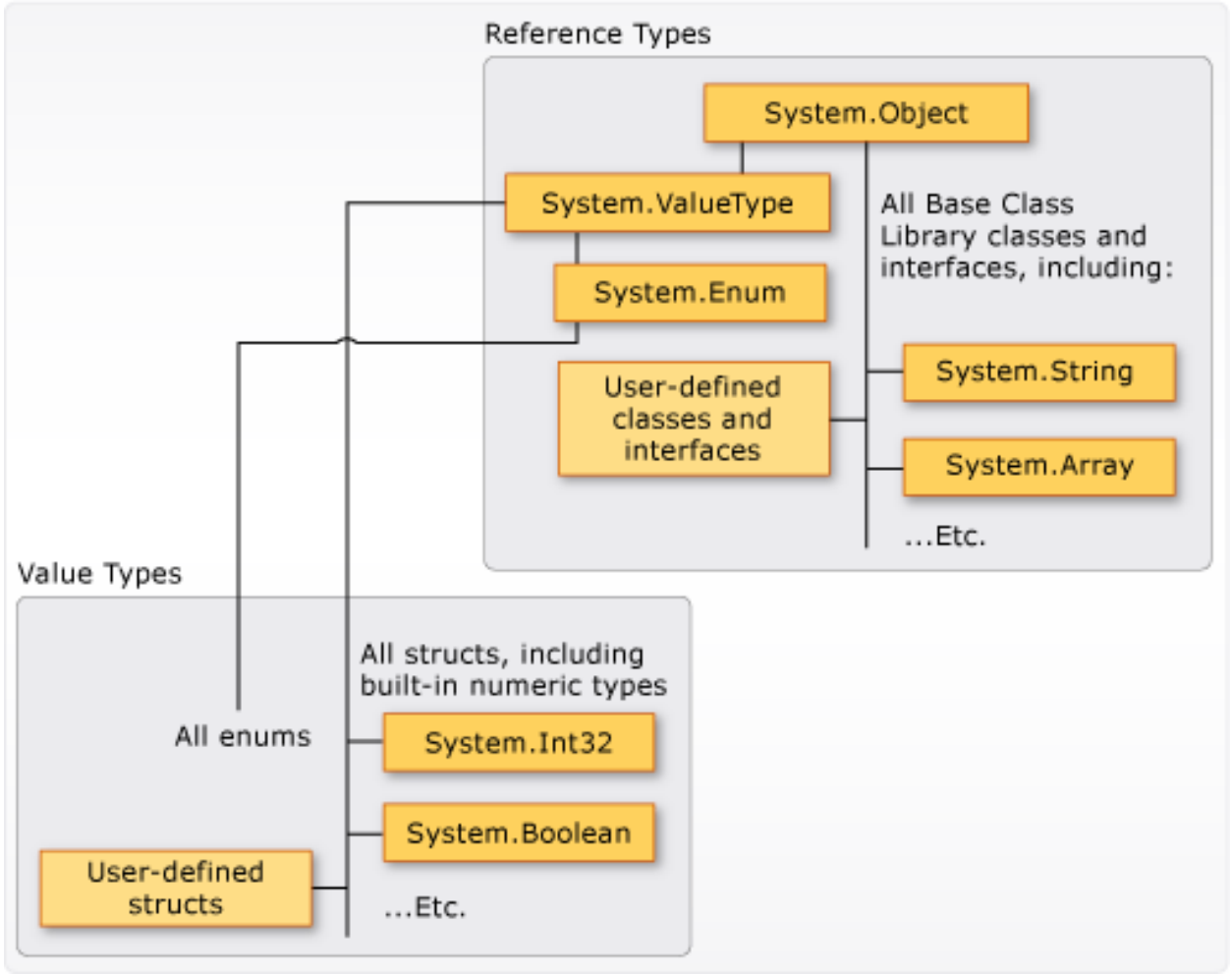
Tipos de dados

- C# é uma linguagem “fortemente tipada” com “análise estática de tipos”
- Cada tipo de dado possui características como:
  - Tamanho de armazenamento
  - Valores iniciais padrão
  - Valores máximos e mínimos
  - Operações e métodos
  - Propriedades

Tipos de dados

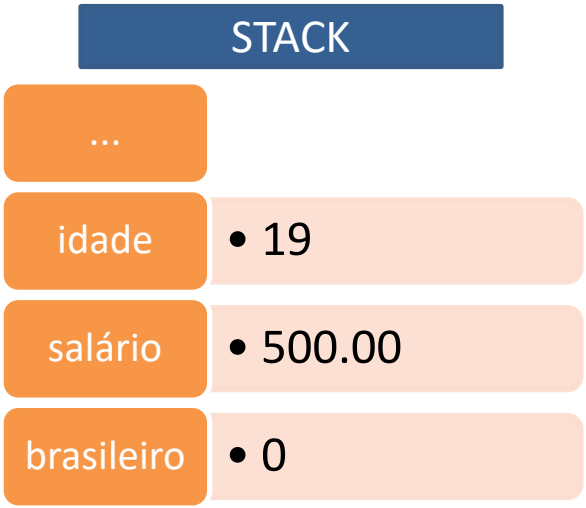
- As diferentes linguagens de programação suportadas pelo .NET (incluindo o C#) tem como base o *Common Type System* (CTS)
- Os tipos de dados no .NET são divididos em dois grupos:
  - Tipos Valor (*value types*)
  - Tipos Referência (*reference types*)

Tipos de dados



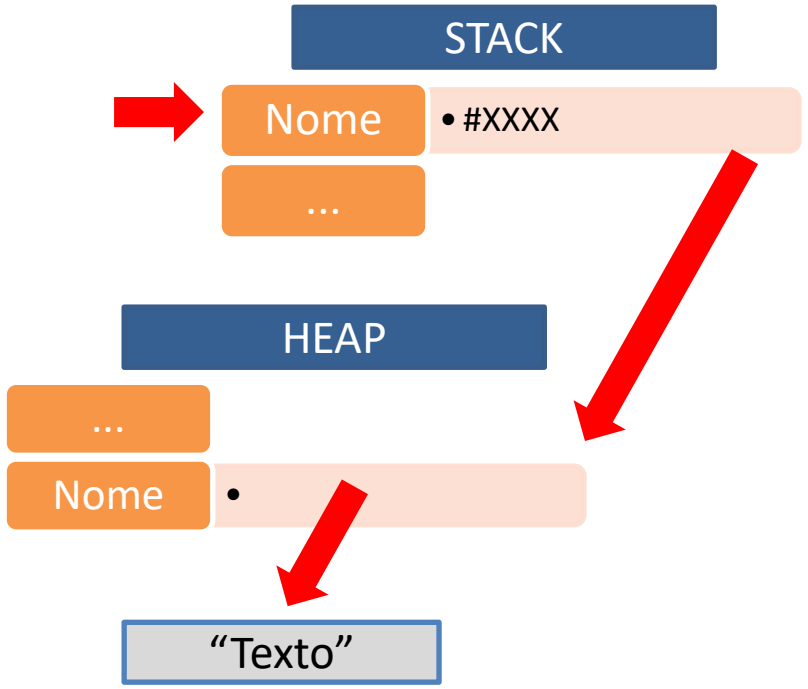
Tipo valor (value type)

- Armazenado na memória Stack.
- Trabalha com dados diretamente.
- Não pode ser nulo.
- Exemplo:
  - Inteiros
  - Decimais
  - Booleanos
  - Estruturas
  - Enumerações



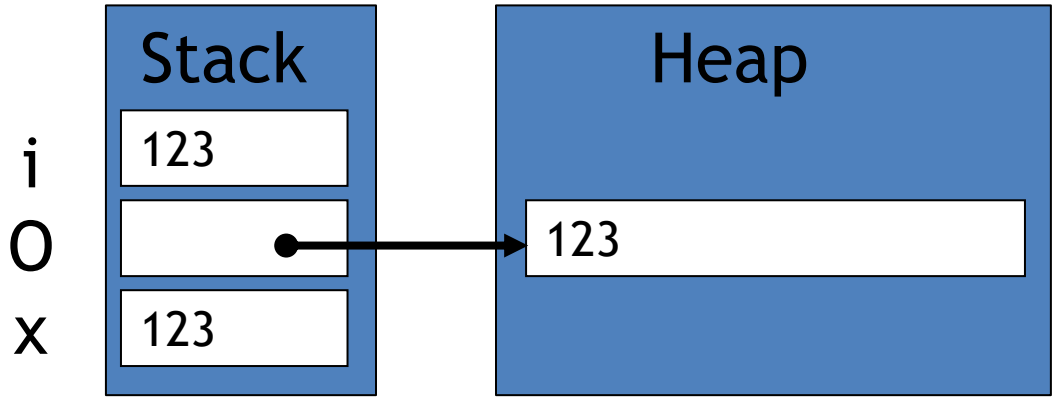
Tipo referência (reference type)

- Contém uma referência a um ponteiro na memória Heap.
- Pode ser nulo
- Exemplo:
  - Classes
  - Interfaces
  - Delegates



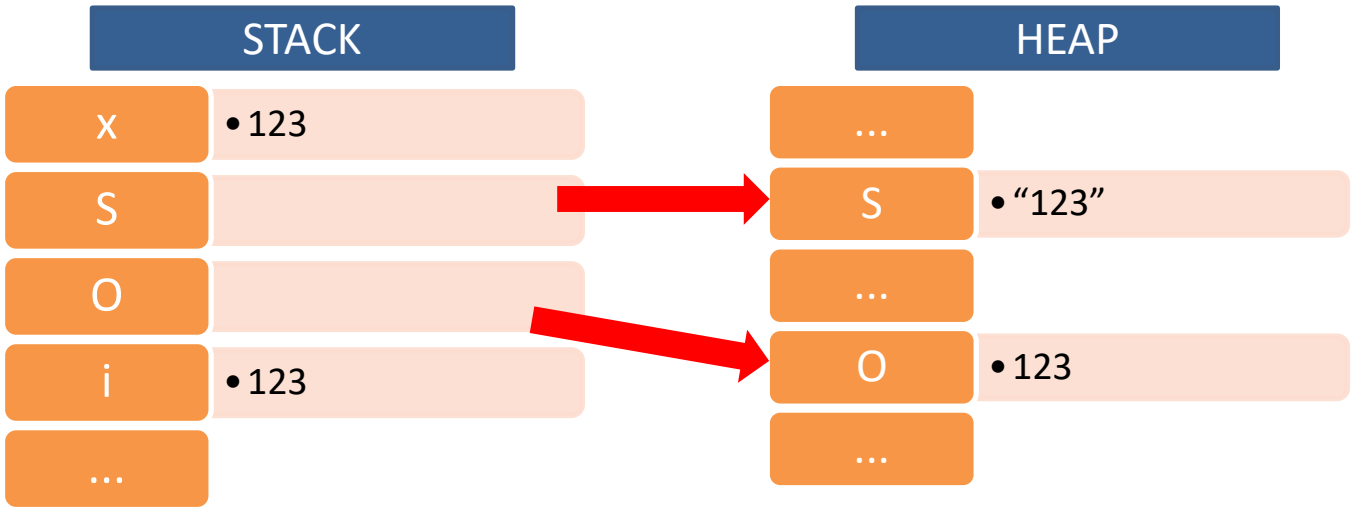
Boxing e unboxing

```
int i = 123;                    // Tipo valor
object O;                       // Tipo referência
O = i;                          // Causa “boxing”
string S;
S = O.ToString()                // Chamada método
int x;
x = (int) O;                    // Faz “unboxing”
```



Boxing e unboxing

```
int i = 123;                    // Tipo valor
object O;
O = i;                          // Causa “boxing”
string S;                       // Tipo referência
S = O.ToString();               // Chamada de método
int x;
x = (int) O;                    // Faz “unboxing”
```



Tipos Anuláveis

- Cuidado!
- .NET possui o suporte a mecanismos de validação de referências a NULL que pode estar habilitado ou não em um projeto
  - Área de linguagens conhecida como “null safety”
  - Está habilitado por padrão nos templates de projetos atuais a partir do .NET 6
  - Visa evitar a ocorrência de exceções *System.NullReferenceException*

Tipos Anuláveis

- Um tipo anulável permite uma referência explícita a NULL
- Tanto tipos valor quanto tipos referência podem ser anuláveis
- Como habilitar:
  - Usar ? logo após a declaração do tipo
  - Ex.: int?, string?
- Novos operadores:
  - Null-forgiving !
  - Null-coalescing ??
  - Null-conditional ?.

Tipos Anuláveis

- Qual o valor de cada variável nas seguintes declarações?

string first;	null
string second = string.Empty;	“”
int third;	0
DateTime date;	DateTime.MinValue

- A variável *second* foi inicializada expliciatemente
- As variáveis *first*, *third*, *date*, recebem o valor *default* associado ao tipo

Tipos Anuláveis

- Qual o valor de cada variável nas seguintes declarações?

int? first;	null
int? second = null;	null
int? third = default;	null
int? fourth = new();	0

- A variável *first* recebe o valor *default* associado ao tipo anulável
- A variáveis *second*, *third* e *fourth* foram inicializadas explicitamente

Tipos Anuláveis

- Qual a intenção sobre o valor de cada variável nas seguintes declarações?

```
string first = string.Empty;
string second;
string? third;
```

- *first* nunca será *null*
- *second* não deve ser *null*, embora tenha o valor inicial *null*
  - Um aviso será emitido pelo compilador se tentar usar a variável sem a inicialização não nula
- *third* pode ser *null*

Operadores

Aritméticos	• +, -, *, /, %
Atribuição	• =, +=, -=, *=, /=, <<=, >>=, &=, ^=, |=
Concatenação	• +
Criação de Objetos	• new
Igualdade e Diferença	• ==, !=
Incremento e Decremento	• ++, --
Lógicos e Bit a bit	• &&, ||, !, &, |, ^, ~
Primários	• typeof, sizeof, checked, unchecked
Relacionais	• <, >, <=, >=, is

Namespaces

- Declarando um namespace

```
namespace Cadastro {
   public class Cliente {}
}
```

- Namespaces encadeados

```
namespace Cadastro.Telas {
   public class TelaCliente {}
}
```

- Instrução Using

```
using System;
using System.Data;
using Cadastro.Telas;
using Pessoa = Cadastro.Cliente;
```

Comentários e regiões

- Criando comentários e regiões:

```
// Comentário de uma linha
/*
   Comentário com
   mais de uma linha
*/
/// <summary>
/// Documentação XML
/// </summary>
private int Atributo;
#region Região
   private int Atributo1;
   private int Atributo2;
#endregion
```

Estruturas de controle de fluxo

- if ... else
  - Comando condicional: executa um bloco de comandos se uma condição for verdadeira.
  - A cláusula else (condição falsa) é opcional.

```
if (idade >= 18)
{
    Console.Write("Autorizado.");
    Console.Write("Sua idade é: " + idade);
}
else if (idade > 15 && idade < 18)
{
    Console.Write("Somente com os pais.");
    Console.Write("Menor de 18 anos.");
}
else
{
    Console.Write("Não autorizado.");
    Console.Write("Menor de 15 anos.");
}
```

Estruturas de controle de fluxo

- switch … case
  - Estrutura de decisão que seleciona um comando com base no valor de uma variável.
  - A cláusula default é opcional.

```
switch (sexo)
{
    case "masculino":
        Console.Write("Homem");
        break;
    case "feminino":
        Console.Write("Mulher");
        break;
    default:
        Console.Write("Não informado");
        break;
}
```

Estruturas de repetição

- for
  - Estrutura de repetição composta por três expressões:
    - Inicialização.
    - Condição de parada.
    - Atualização.

```
for (int i = 0; i < 10; i++)
{
    Console.Write(i);
    Console.Write("\n");
}
```

Estruturas de repetição

- while
  - Estrutura de repetição que realiza as operações indicadas enquanto a condição especificada for verdadeira.

```
int i = 0;
while(i < 10)
{
    Console.Write(i);
    Console.Write("\n");
    i++;
}
```

Estruturas de repetição

- do ... while
  - Estrutura de repetição semelhante à anterior, porém as condições são verificadas ao final da execução.
  - As operações especificadas são executadas pelo menos uma vez.
  - Necessita do caractere “;” ao final da estrutura.

```
int i = 0;
do
{
    Console.Write(i);
    Console.Write("\n");
    i++;
}
while (i < 10);
```

Array

- System.Array é um tipo referência que permite o armazenamento de uma coleção de valores de um mesmo tipo de dados.
- Array é indexado a partir de zero (0).
- Não pode ter seu tamanho alterado depois de instanciado.

Array

- Para declarar um Array, basta adicionar um par de colchetes logo após a declaração do tipo dos elementos individuais

```
int[] meuArrayDeInteiros;
string[] meuArrayDeStrings;
```

Array

- Instanciando arrays

```
int[] codigos = new int[5];
string[] nomes = new string[100];
object[] produtos = new object[50];
int[] pedidos = {1, 4, 6, 8, 10, 68, 90, 98, 182, 500};
```

Array

- Preenchendo um array

```
nomes[0] = "José";
nomes[1] = "João";
```

Array

- Arrays podem ser:
  - Unidimensionais
  - Multidimensionais
  - Jagged

Array

- Unidimensionais

```
int[] codigos = new int[5];
codigos[0] = 1;
```

```
int[] codigos = {1,3,6,7,8};
```

Array

- Multidimensionais

```
int[,] codigos = new int[2,2];
codigos[0,0] = 11;
```

```
int[,] codigos =
   {{11,42},{35,44}};
```

Array

- Jagged

```
int[][] codigos = new int[2][];
codigos[0] = new int[2];
codigos[0][0] = 11;
```

```
int[][] codigos = { new int[]{11,42}, new int[]{35,44} };
```

Estruturas de repetição

- foreach
  - Esta estrutura de repetição é uma variação do for.
  - Especifica uma variável auxiliar e a coleção ou array cujos elementos serão percorridos.

```
int[] i = { 1, 3, 5, 7, 9 };
foreach (int j in i)
{
    Console.Write(j);
    Console.Write("\n");
}
```

Coleções

- São classes usadas para agrupar e gerenciar objetos relacionados e que permitem armazenar, buscar e interagir com estes objetos
- As coleções possuem mais funcionalidades do que um array, facilitando sua utilização

Coleções

- O namespace *System.Collections* contêm diversos tipos de coleções não-genéricas

Nome	Descrição
ArrayList	Uma simples coleção de objetos redimensionável e baseada em index.
SortedList	Uma coleção de pares nome/valor ordenada por chave.
Queue	Uma coleção de objetos First-in, First-out.
Stack	Uma coleção de objetos Last-in, First-out.
...	...

Tipos genéricos

- Genéricos são construções do sistema de tipos do .NET que permitem a construção de novos tipos com flexibilidade de tipagem
- Introduzem o conceito de tipos parametrizados

Tipos genéricos

- Pode ser aplicado em:
  - Classes
  - Interfaces
  - Métodos
  - Structs
  - Delegates

Tipos genéricos

- Vantagens:
  - Diminui a necessidade do uso de sobrecarga
  - Permitem criar estruturas de dados sem se restringir a um tipo específico
    - É o exemplo de uso mais utilizado
    - Evita erros de conversão em tempo de run-time de e para Object
  - Maior desempenho
    - Evita boxing / unboxing
  - Verifica o tipo em tempo de compilação

Coleções genéricas

- Disponibilizadas no namespace *System.Collections.Generic*
  - É o tipo de coleções mais recomendado
- Principais coleções:
  - List, LinkedList, SortedList
  - Dictionary, SortedDictionary
  - KeyedCollection
  - Queue
  - Stack

Coleções genéricas

System.Collections	System.Collections.Generics
ArrayList	List<>
Queue	Queue<>
Stack	Stack<>
Hashtable	Dictionary<>
SortedList	SortedList<>
ListDictionary	Dictionary<>
HybridDictionary	Dictionary<>
OrderedDictionary	Dictionary<>
SortedDictionary	SortedDictionary<>
NameValueCollection	Dictionary<>

Coleções genéricas

System.Collections	System.Collections.Generics
StringCollection	List<String>
StringDictionary	Dictionary<String>
N/A	LinkedList<>

- Todas as classes apresentadas acima possuem funcionalidades e métodos semelhantes a sua correspondente no namespace *System.Collections*, exceto a classe *LinkedList<>*, que é exclusiva do namespace *System.Collections.Generic*.

Enumerações

- Definindo tipos enumerados

```
// Declarando
enum DiasUteis
{
   Segunda, Terca, Quarta, Quinta, Sexta
}

...
// Instanciando
DiasUteis du = DiasUteis.Sexta;

// Imprime "Sexta"
Console.Write(du);
```

Classes

- Classes definem tipos de dados do tipo referência
- Determina um conjunto de objetos com:
  - propriedades semelhantes
  - comportamentos semelhantes
  - relacionamentos comuns com outros objetos

Classes

- Membros das classes
  - Constantes, campos, métodos, propriedades, indexadores, eventos, operadores, construtores, finalizadores
  - Tipos aninhados
  - Membros “de instância” e “de classe”

Modificadores de acesso

- Modificadores de acesso são utilizados para definir níveis de acesso a membros das classes

Declaração	Definição
public	Acesso ilimitado
private	Acesso limitado à classe e seus membros
internal	Acesso limitado ao código no mesmo assembly (exe ou dll)
protected	Acesso limitado à classe, seus membros e a tipos derivados da mesma
protected internal	Acesso limitado à classe, classes derivadas ou classes do mesmo assembly
private protected	Acesso limitado à classe, classes derivadas no mesmo assembly

Instâncias

- Objetos são gerados a partir de classes
- Uma classe define as propriedades e o comportamento dos objetos gerados por ela
- Todo objeto é uma instância de uma classe

Classes - campos

- Definindo uma classe e seus campos

```
public class Cliente {
  private string nome;
  private decimal limiteCredito;
  private uint   clienteID;
}
```

- Instanciando uma classe

```
Cliente proximoCliente = new Cliente();
```

Classes - constantes

- Definindo uma classe e campos constantes

```
public class Calendario {
  public const int meses = 12;
}
```

- Acessando constantes (observe que é um membro de classe)

```
int valor = Calendario.meses;
```

Classes - métodos

- Métodos representam as ações associadas à classe

```
public void AumentarLimite(decimal val) {
     limiteCredito += val;
}
```

- Chamando um método

```
proximoCliente.AumentarLimite(100M);
```

Classes - métodos

- Os parâmetros de um método podem receber um modificador que indica a direção do fluxo de dados
  - Entrada
  - Saída
  - Entrada/saída

[nenhum]	Se não existe modificador, assume que é parâmetro de entrada passado por valor.
out	Parâmetro de saída. Seu valor será atribuído pelo próprio método, não necessitando de inicialização prévia.
ref	Parâmetro de entrada/saída. Seu valor pode ser atribuído antes da chamada do método ou sofrer alteração pelo método. Usado para passar tipos valor “por referência”
params	Permite receber um número variável de parâmetros através de um array.

Classes - métodos

- Parâmetro de saída

```
public void Adicionar(int x, int y, out int r) {…}
Adicionar(1, 2, out resultado);
```

- Parâmetro de entrada/saída

```
public void ParaMaiuscula(ref string s) {…}
ParaMaiuscula(ref frase);
```

- Parâmetros variáveis

```
public void MostrarLista(params int[] lista) {…}
int[] array = new int[3] {1,2,3};
MostrarLista(array);
MostrarLista(1,2,3);
MosttarLista(1,2,3,4,5);
```

Classes - propriedades

- Propriedades representam dados/informações associadas a uma instância

```
private string nome;
public string Nome {
        get {   return nome; }
        set {   nome = value; }
}
```

- Acessando propriedades

```
proximoCliente.Nome = "Microsoft";
```

Classes - propriedades

- Get e Set auto-implementados:

```
public string Nome {
        get;set;
}
```

- Acessando propriedades:

```
proximoCliente.Nome = "Microsoft";
```

Classes - construtores

- Construtores são métodos especiais que implementam as ações necessárias para inicializar um objeto
  - Tem o mesmo nome da classe
  - Não possuem tipo de retorno (nem *void*)
  - Podem ou não possuir parâmetros

```
public Cliente(string n, uint i) {
      nome = n;
      clienteID = i;
}
```

Classes - indexadores

- Indexadores são propriedades parametrizadas por índices
  - Em C# o indexador tem nome fixo *this*

```
public string this[int i] {
        get {   … }
        set {   … }
}
```

```
umaString = objetoComIndexador[3];
```

Classes - sobrecarga

- Chama-se de sobrecarga de métodos (*overloading*) o ato de criar diversos métodos com o mesmo nome que se diferenciam pela lista de parâmetros
- Métodos com mesmo nome, mas com tipo, quantidade ou ordenação de parâmetros diferentes, são considerados métodos diferentes

Classes - sobrecarga

- Exemplo: sobrecarga de construtor

```
public class Data
{
  private int dia, mes, ano;
  public Data(int d, int m, int a) {
    dia = d;
    mes = m;
    ano = a;
  }
  public Data(Data d) : this(d.dia, d.mes, d.ano){
  }
}
```

Classes - membros estáticos

- Membros estáticos são acessados através da própria classe, sem a necessidade de instanciar um objeto
- São declarados com o modificador *static*

Classes - membros estáticos

- Definindo membros estáticos

```
public class SuperComputador {
   private static int Resposta = 42;
   public static string ObterPergunta(){...}
}
```

- Acessando membros estáticos

```
Console.Write(SuperComputador.ObterPergunta());
```

Eventos e Delegates

- Conceitos:
  - Evento: ação que pode ser gerenciada/manipulada através de código
    - São baseados principalmente nos tipos *EventHandler* e *EventArgs*
  - Delegate: membro da classe responsável por “delegar” as ações correspondentes a ocorrência de um evento ao(s) manipulador(es) de eventos correspondentes
    - É um tipo referência que suporta referências para métodos (os manipuladores de um evento associado)
  - Manipulador de evento: método responsável pela execução de ações em reação a ocorrência de um evento

Eventos e Delegates

Cinco passos para se trabalhar com eventos

- Passo 1: declarar o *delegate* contendo a assinatura do manipulador de evento correspondente ao evento

```
public delegate void FazAlgoDelegate(int x);
```

- Passo 2: declarar o evento (deve ser do mesmo tipo do delegate correspondente)

```
public class UmaClasse
{
    public event FazAlgoDelegate UmEvento;
}
```

Eventos e Delegates

- Passo 3: disparar o evento na chamada de algum método da classe

```
public class UmaClasse
{
    ...
    public void MetodoEvento(int x) {
        if(UmEvento != null) UmEvento(x);}
}
```

- Passo 4: assinar o evento indicando o manipulador de eventos do mesmo através de uma instância de delegate

```
UmaClasse obj = new UmaClasse();
obj.UmEvento += new FazAlgoDelegate(ManipuladorEvento);
```

Eventos e Delegates

- Passo 5: implementar o manipulador de evento (deve respeitar a mesma assinatura definida pelo delegate do evento)

```
public void ManipuladorEvento(int x)
{
    Console.Write("Evento processado: valor="+x);
}
```

C# - expressões lambda

- Construção semanticamente equivalente a um *delegate* anônimo via operador =>

  (parâmetros de entrada) => corpo da expressão

```
public static int Soma(int a, int b) {return a+b;}
Func<int,int,int> umDelegate = Soma;
```

```
Func<int,int,int> metodoAnonimo = delegate(int a, int b) {
    return a+b;
};
```

```
Func<int,int,int> lambda = (a,b) => (a+b);
```

Herança

- Herança é uma relação de especialização entre classes
- A idéia central de herança é que novas classes são criadas a partir de classes já existentes
  - Subclasse herda de uma Superclasse
  - Subclasse é mais específica que a Superclasse
- Herança origina uma estrutura em árvore

Herança

- Para definir a herança de classes em C# utiliza-se um “:” seguido do nome da superclasse
- C# suporta herança simples de classes

```
public class Classe : SuperClasse {
      …
}
```

Herança

- Ao definir os construtores de uma subclasse:
  - O construtor deve obrigatoriamente chamar o construtor da classe base para inicializar os atributos herdados
  - Caso um construtor não referencie o construtor da classe base, C# automaticamente referencia o construtor vazio da classe base
  - O construtor pode referenciar explicitamente um construtor da classe base via a palavra-chave base após a assinatura do construtor da subclasse e da marca “:”

Sobrescrita de métodos

- Uma subclasse pode sobrescrever (do inglês *override*) métodos da superclasse
  - Sobrescrita permite completar ou modificar um comportamento herdado
  - Quando um método é referenciado em uma subclasse, a versão escrita para a subclasse é utilizada, ao invés do método na superclasse
  - Em C#, um método que sobrescreve um método herdado é marcado pela palavra-chave override

Sobrescrita de métodos

- Um método de uma classe, que pode ser sobrescrito em uma subclasse, deve ser marcado pela palavra-chave virtual
- O método herdado pode ser referenciado através da construção base.nome_método

```
public class SuperClasse {
  public virtual void Metodo(){…}
}
public class Classe : SuperClasse {
  public override void Metodo() {…}
}
```

Polimorfismo

- Polimorfismo é a capacidade de assumir formas diferentes

- C# permite a utilização de variáveis polimórficas
  - Uma mesma variável permite referência a objetos de tipos diferentes
  - Os tipos permitidos são de uma determinada classe e todas as suas subclasses

Polimorfismo

- Uma variável do tipo da superclasse pode armazenar uma referência da própria superclasse ou de qualquer uma de suas subclasses

```
public class Classe : SuperClasse {
      …
}

SuperClasse obj;
obj = new Classe();
```

Operadores de polimorfismo

- **IS** e **AS**

```
if (computador is Produto)
{
   // ações
}
```

```
Produto produto = computador as Produto;

if (produto != null)
{
   Fornecedor fornecedor = produto.Fornecedor;
}
```

Polimorfismo

- Em C# podemos utilizar métodos polimórficos
  - Uma mesma operação pode ser definida em diversas classes de uma hierarquia.
    - cada classe oferece sua própria implementação utilizando o mecanismo de sobrescrita de métodos

Classes abstratas

- Em uma hierarquia de classe, quanto mais alta a classe, mais abstrata é sua definição
  - Uma classe no topo da hierarquia pode definir apenas o comportamento e atributos que são comuns a todas as classes
  - Em alguns casos, a classe nem precisa ser instanciada diretamente e cumpre apenas o papel de ser um repositório de comportamentos e atributos em comum
- É possível definir classes, métodos e propriedades abstratas em C#

Classes abstratas

- Marca-se a classe com a palavra-chave *abstract*

```
public abstract class Funcionario()
{
  public abstract decimal CalcularSalario();
  public abstract string Codigo {get; set;}
}
```

Herança – palavras-chave

ABSTRACT	• Indica uma classe, método ou propriedade que não admite instâncias diretamente.
OVERRIDE	• Indica uma redefinição em uma classe derivada.
VIRTUAL	• Indica um elemento da classe base que pode ser redefinido.
THIS	• Indica um elemento da própria classe.
BASE	• Indica um elemento da classe base.
SEALED	• Indica uma classe que não admite derivadas.

Modificadores de classes

- Public: permite que a classe seja acessada por qualquer assembly.
- Sealed: não permite que a classe seja herdada.
- Partial: permite que a classe tenha seu escopo dividido em dois arquivos.
- Static: especifica que a classe somente tem membros estáticos. Não pode ser instanciada.
- Abstract: define moldes para classes filhas. Não pode ser instanciada.

Modificadores de membros

- Public: permite que os membros das classes sejam acessados por qualquer outro escopo.
- Private: acesso restrito ao escopo da classe.
- Protected: acesso restrito a classe e as derivadas.
- Internal: permite acesso somente por classes do mesmo assembly.
- Static: permite acesso, sem necessidade do objeto ser instanciado.
- Abstract: são métodos de classes Abstract que não possuem implementação.
- Virtual: permite que os métodos sejam sobrescritos por classes filhas.
- Readonly: limita acesso a somente leitura aos atributos da classe.

Interfaces

- Interfaces podem ser utilizadas para separar a especificação do comportamento de um objeto de sua implementação concreta
- Dessa forma a interface age como um contrato, o qual define explicitamente quais métodos uma classe deve obrigatoriamente implementar
  - Por exemplo, suponha a necessidade de implementação da estrutura de dados Pilha
    - Toda pilha deve possuir as operações empilha(), desempilha(), estaVazia()
    - Mas a pilha pode ser implementada com array, lista encadeada, etc

Interfaces

- Existem dois motivos básicos para fazer uso de interfaces:
  - Uma interface é como um contrato que determina o que deve fazer parte de suas classes derivadas;
  - Bibliotecas padronizadas de interfaces uniformizam a construção de projetos.
- Uma interface informa apenas quais são o nome, tipo de retorno e os parâmetros dos métodos.
  - A forma como os métodos são implementados não é preocupação da interface.
  - A interface representa o modo como você quer que um objeto seja usado.

Interfaces

- Declarando interfaces:
  - Uma interface é declarada de forma semelhante a uma classe
  - Utiliza-se a palavra-chave *interface* ao invés de *class*
  - Em C#, interfaces podem conter métodos, propriedades, indexadores e eventos
  - Não é possível fornecer modificadores para os membros da interface
    - São implicitamente públicos e abstratos

Interfaces

- Restrições importantes:
  - Uma interface não permite a presença de atributos
  - Uma interface não permite construtores/destrutores
    - Não é possível instanciar uma interface
  - Não é possível fornecer modificadores para os membros da interface
  - Não é possível aninhar declaração de tipos dentro de uma interface
  - Interfaces somente podem herdar de outras interfaces

Interfaces

- Declarando uma interface:

```
interface IPilha {
  void Empilhar(object obj);
  object Desempilhar();
  object Topo{get;}
}
```

Interfaces

- Implementando interfaces:
  - Como interfaces são compostas de métodos abstratos, esses métodos deverão ser implementados por alguma classe concreta
  - Logo, dizemos que uma interface é implementada por uma classe
  - Utiliza-se a mesma notação de herança
  - A classe deverá implementar todos os métodos listados na interface
    - A implementação deve ser pública, não estática e possuir a mesma assinatura
  - Uma classe pode implementar diversas interfaces

Interfaces

- Implementando uma interface:

```
public class PilhaArray : IPilha {
  private object[] elementos;
  public void Empilhar(object obj){...}
  public object Desempilhar(){...}
  public object Topo{
      get {...}
  }
  ...
}
```

Interfaces

- Implementação explícita de interfaces:
  - Se uma classe implementa duas interfaces que contêm um membro com a mesma assinatura, a mesma implementação será utilizada para as duas interfaces
    - Esta característica pode tornar o código inconsistente
  - C# permite implementar explicitamente um método de uma interface agregando o nome da interface antes do nome do método
    - Como consequência, os métodos somente poderão ser chamados via uma variável do tipo da interface adequada

Interfaces

- Implementando explicitamente uma interface:

```
interface IUmaInterface {
   void metodo();
}
interface IOutraInterface {
   void metodo();
}
public class MinhaClasse : IUmaInterface, IOutraInterface
{
   public void IUmaInterface.metodo(){...}
   public void IOutraInterface.metodo(){...}
}
```

Polimorfismo

- Quando declaramos uma variável como sendo do tipo de uma interface, essa variável irá aceitar qualquer objeto de uma classe que implemente essa interface
- Dessa maneira, temos acessos aos métodos definidos na interface de forma independente do tipo de objeto que estamos utilizando

Polimorfismo

```
interface IMinhaInterface {
      …
}

public class Classe : IMinhaInterface {
      …
}

MinhaInterface obj;
obj = new Classe();
```

Interfaces do .NET

- No ambiente .NET temos uma grande quantidade de interfaces pré-definidas. Por exemplo:
  - IComparable e IComparer para a ordenação de objetos.
  - IEnumerable e IEnumerator para implementar a operação foreach.
  - ICloneable para permitir a criação de cópia de objetos
  - IFormattable para definir cadeias de caracteres formatadas.
  - IDataErrorInfo para associar mensagens de erros a uma classe.

Interfaces do .NET

- IComparable:
  - Interface para comparação de valores segundo alguma ordem parcial
  - Define o método CompareTo() que deve retornar um valor inteiro com o resultado da comparação
    - Menor que zero – se a instância atual é menor que o valor do parâmetro
    - Zero – se a instância atual é igual ao valor do parâmetro
    - Maior que zero – se a instância atual é maior que o valor do parâmetro

Interfaces do .NET

- IComparer:
  - Permite diferentes algoritmos de comparação de valores
  - Define o método Compare() que recebe dois objetos e deve retornar um valor inteiro com o resultado da comparação
    - Menor que zero – se o primeiro objeto for menor que o segundo objeto, de acordo com o algoritmo implementado
    - Zero – se os objetos forem iguais
    - Maior que zero – se o primeiro objeto for maior que o segundo objeto

Estruturas

- Estruturas são tipos por valor, que podem conter:
  - Construtor obrigatoriamente com parâmetros
  - Constantes
  - Atributos
  - Métodos
  - Propriedades
- Uso recomendado para representar objetos leves e/ou que eventualmente podem constituir arrays de grande dimensão.
- Podem ser instanciados sem a utilização de *new*
- Não podem ser herdados de outra estrutura ou classe, porém podem implementar interfaces.

Estruturas

- Exemplo de uma estrutura:

```
struct Circulo {
    private int _raio;          // Atributo
    public double Circunferencia      // Propriedade
    { get { return 2 * _raio * Math.PI; } }
                // Regra específica para retornar  um valor.
    public Circulo(int raio)  // Construtor com um argumento
    { this._raio = raio; }    // Atribuição do valor do argumento
}                             // para o atributo do objeto.
// Instancia de uma estrutura.
Circulo meuCirculo = new Circulo(10);
// Imprime o valor de uma propriedade
Response.Write(meuCirculo.Circunferencia);
}
```

Estruturas

- Exemplo de sobrecarga de métodos:

```
... Dentro da mesma estrutura do exemplo anterior:

public void DiminuirRaio(){   // Método simples sem argumentos
    if (_raio > 1)
        _raio--;
}

public void DiminuirRaio(int valor){// Overload do método anterior
    if (_raio - valor > 1)            // com um argumento
        _raio -= valor;
}

... Dentro do evento Page_Load

meuCirculo.DiminuirRaio();    // Chamando o método sem argumentos
meuCirculo.DiminuirRaio(2);   // Chamando o overload
                              // do método anterior
```