# Unidade II

## A LINGUAGEM DE PROGRAMAÇÃO C#

Nesta unidade, veremos os aspectos de umas das linguagens de programação mais utilizadas no mundo: a C# .NET.

Veremos sua estrutura e seu funcionamento básico, de modo que sejamos capazes de implementar pequenos programas e testarmos os aspectos da POO na prática.

## 3 CONHECENDO A LINGUAGEM DE PROGRAMAÇÃO C#

### 3.1 Visão geral da plataforma Microsoft .NET

A .NET é uma plataforma de desenvolvimento da Microsoft que tem como foco principal o desenvolvimento de serviços Web XML. A ideia central de um serviço Web XML, ou simplesmente *web service*, consiste em permitir que as aplicações, sejam elas da *web*, do *desktop* ou ainda de *middleware*, se comuniquem e troquem dados de forma simples e transparente, independentemente do sistema operacional ou da linguagem de programação. Sua proposta é envolver linguagens de programação, compiladores, modelo de objetos etc. necessários para que consiga englobar de uma forma completamente integrada todos esses requisitos (LIMA; REIS, 2002).

Deste modo, a plataforma .NET possibilita a integração de diversas linguagens de programação, tais como C#, VB.NET (Visual Basic .NET), JScript.NET, Managed C++, HTML e conexão com banco de dados (SQL).

Resumidamente, a plataforma .NET possui as seguintes características (LIMA; REIS, 2002):

- **Independência de linguagem de programação**, o que permite a implementação do mecanismo de herança, o controle de exceções e a depuração entre linguagens de programação diferentes.

- **Reutilização de código legado**, o que implica reaproveitamento de código escrito usando outras tecnologias como COM, COM+, ATL, DLLs e outras bibliotecas existentes.

- **Tempo de execução compartilhado**, o *runtime* de .NET é compartilhado entre as diversas linguagens que a suportam, o que quer dizer que não existe um *runtime* diferente para cada linguagem que implementa a .NET.

- **Sistemas autoexplicativos e controle de versões**: cada peça de código .NET contém em si mesma a informação necessária e suficiente de forma que o *runtime* não precise procurar no

registro do Windows mais informações sobre o programa que está sendo executado. O *runtime* encontra essas informações no próprio sistema em questão e sabe qual a versão a ser executada, sem acusar aqueles velhos conflitos de incompatibilidade ao registrar DLLs no Windows.

- **Simplicidade na resolução de problemas complexos**.

A plataforma .NET é dividida de forma modular, de modo a garantir integridade e robustez na execução das aplicações. Tais módulos ou componentes são: CLR (Commom Language Runtime), CTS (Common Type System), CLS (Common Language Specification) e BCL (Base Class Library). Sua hierarquia pode ser vista na figura a seguir:

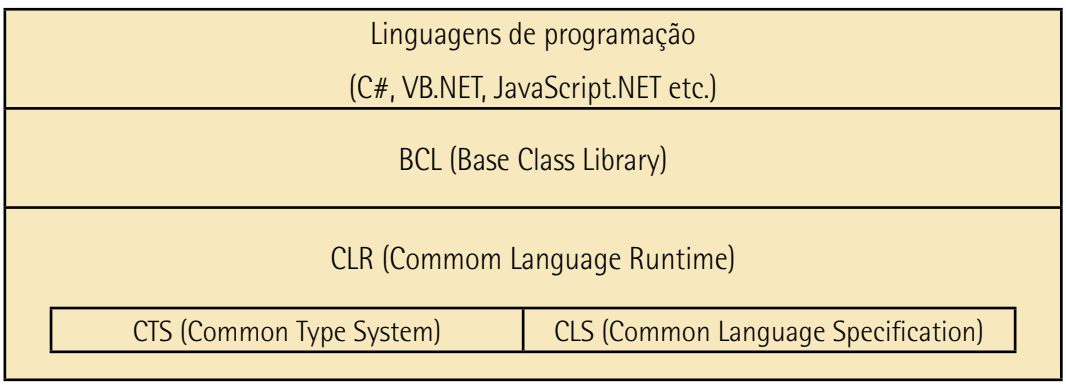


Figura 7 – Arquitetura .NET

O CLR, ou tempo de execução compartilhado, é o ambiente de execução das aplicações .NET. As aplicações .NET não são aplicações Win32 propriamente ditas (apesar de executarem no ambiente Windows), razão pela qual o *runtime* Win32 não sabe como executá-las. O Win32, ao identificar uma aplicação .NET, dispara o *runtime* .NET, que, a partir desse momento, assume o controle da aplicação no sentido mais amplo da palavra. Isso porque, dentre outras coisas, é ele quem vai cuidar do gerenciamento da memória via um mecanismo de gerenciamento de memória chamado Garbage Collector (GC) ou coletor de lixo, sobre o qual falaremos mais tarde. Esse gerenciamento da memória torna os programas menos suscetíveis a erros. Além disso, o CLR, como seu próprio nome diz, é compartilhado e, portanto, não temos um *runtime* para VB.NET, outro para C# etc., e sim o mesmo para todos (LIMA; REIS, 2002).

O CTS, ou Sistema Comum de Tipos, que também faz parte do CLR, define os tipos suportados por .NET e as suas características. Cada linguagem que suporta .NET tem de, necessariamente, suportar esses tipos. Embora a especificação não demande que todos os tipos definidos no CTS sejam suportados pela linguagem, esses tipos podem ser um subconjunto (classe filho) do CTS ou ainda um superconjunto (classe pai), conforme Lima e Reis (2002).

O CLS, ou Especificação Comum da Linguagem, é um subconjunto do CTS e define um conjunto de regras que qualquer linguagem que implemente a .NET deve seguir, a fim de que o código gerado resultante da compilação de qualquer peça de *software* escrita na referida linguagem seja perfeitamente entendido pelo *runtime* .NET. Seguir essas regras é um imperativo porque, caso contrário, um dos grandes ganhos do .NET, que é a independência da linguagem de programação e a sua interoperabilidade, fica

comprometido. É importante entender esse conceito para não pensar que o código desenvolvido em C# não pode interagir com o código desenvolvido em VB ou outras linguagens, porque mesmo estas sendo diferentes, são todas compatíveis com o CLS.

Ainda conforme Lima e Reis (2002), a BCL (Biblioteca de Classes Base) oferece ao desenvolvedor uma biblioteca consistente de componentes de *software* reutilizáveis que não apenas facilitam, mas também aceleram o desenvolvimento de sistemas. Na BCL, encontramos classes que contemplam desde um novo sistema de janelas a bibliotecas de entrada/saída, gráficos, *sockets*, gerenciamento da memória etc. Essa biblioteca de classes é organizada hierarquicamente em uma estrutura conhecida como *namespace*. Ao desenvolver um componente de *software* reusável, este precisa ser estruturado em um *namespace* para que possa ser usado a partir de outro programa externo.

O quadro a seguir exemplifica os tipos de classes existentes na BCL:

**Quadro 2 – Hierarquia de classes .NET**

| Exemplos de classes (*namespace*) .NET | |
|---|---|
| System | Contém algumas classes de baixo nível usadas para trabalhar com tipos primitivos, operações matemáticas, gerenciamento de memória etc. |
| System.Collections | Pensando em implementar suas próprias pilhas, filhas, listas encadeadas? Elas já foram implementadas e se encontram aqui. |
| System.Data, System.Data.Commom, System.Data.OleDb, System.Data.SqlClient | Aqui você vai encontrar tudo o que é necessário para lidar com bases de dados e, como é de se esperar, você encontrará ADO.NET aqui. |
| System.Diagnostics | Log de Event, medição de *performance*, classes para gerenciamento de processos, depuração e mais você poderá encontrar neste *namespace*. |
| System.Drawing | A .NET oferece uma biblioteca de componentes para trabalhar com gráficos, chamada GDI+, que se encontra neste *namespace*. |
| System.IO | Biblioteca para lidar com entrada e saída, gerenciamento de arquivos etc. |
| System.NET | Aqui você encontra bibliotecas para programação de redes, *sockets* etc. |
| System.Reflection | Em .NET você pode gerar código em tempo de execução, descobrir tipos de variáveis etc. As bibliotecas necessárias para isso encontram-se neste *namespace*. |
| System.Runtime.InteropServices, System.Runtime.Remoting | Fornecem bibliotecas para interagir com código não gerenciado. |
| System.Security | Criptografia, permissões e todo o suporte ao qual .NET oferece a segurança você encontra aqui. |
| System.Threading | Bibliotecas necessárias para o desenvolvimento de aplicações *multithread*. |
| System.Web | ASP.NET, *web services* e tudo o que tenha a ver com *web* pode ser encontrado aqui. |
| System.Windows.Forms | Bibliotecas para o desenvolvimento de aplicações Windows tradicionais. |
| System.XML | Bibliotecas que permitem a interação com documentos XML. |

Fonte: Lima; Reis (2002).

Um última informação importante que devemos ter em relação à plataforma .NET se refere à MSIL (Microsoft Intermediate Language).

A MSIL – ou simplesmente IL – é a linguagem intermediária para a qual é interpretado qualquer programa .NET, independentemente da linguagem em que este for escrito. Essa tradução é feita para código intermediário (como em Java com os *byte codes*) sintaticamente expresso na IL. Por sua vez, qualquer linguagem .NET compatível, na hora da compilação, gerará código IL e não código *assembly* específico da arquitetura do processador em que a compilação do programa é efetuada, conforme aconteceria em C++ ou Delphi, por exemplo. Isso acontece para garantir duas coisas: a independência da linguagem e a independência da plataforma (arquitetura do processador) (LIMA; REIS, 2002).

## 3.2 Instalação da linguagem C#

Antes de começar a estudar especificamente a linguagem C#, é recomendável a instalação do programa que irá auxiliar no desenvolvimento dos exemplos da orientação a objetos: o Microsoft Visual C# Express Edition.

**Saiba mais**

A versão 2010 desse programa pode ser encontrada no *site* a seguir:

DOWNLOADS do Visual Studio. [s.d.]. Disponível em: <http://www.visualstudio.com/pt-br/downloads/download-visual-studio-vs#DownloadFamilies_2>. Acesso em: 23 out. 2014.

O programa também pode ser encontrado dentro do pacote Visual Studio da Microsoft. Usualmente, a licença vence em trinta dias, mas basta fazer o registro para que seja enviada a chave do produto que liberará definitivamente o uso do programa.

**Observação**

Os exemplos deste livro-texto foram desenvolvidos utilizando o Microsoft Visual C# 2010 Express. Entretanto, o leitor poderá optar por outra versão mais adequada ao seu estudo ou optar por utilizar o Microsoft Visual Studio 2010 ou superior.

### 3.2.1 Iniciando um novo projeto no C#

Uma vez instalado o programa, os primeiros passos para o uso são os que apresentaremos a seguir.

Em primeiro lugar, procure o ícone do Microsoft Visual C# no Windows. Se o sistema operacional for o Windows 7, ele se apresentará como na figura a seguir:


Figura 8 – Ícone no Menu Iniciar no Windows 7

Caso o sistema operacional seja o Windows 8, ela aparecerá da seguinte maneira:


Figura 9 – Ícone na Tela Iniciar do Windows 8

Para iniciar um novo projeto, vá em **File/New Project**:

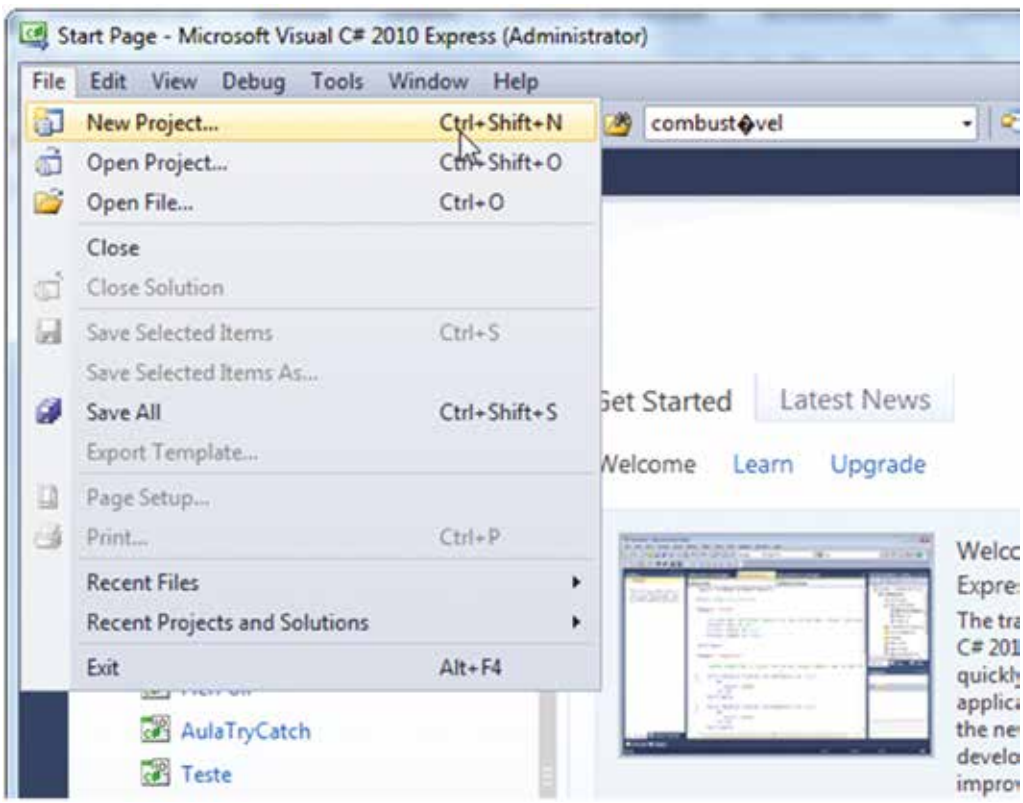


Figura 10 – Iniciando um novo projeto

Aparecerá então a tela para escolher o tipo de projeto (neste curso, será sempre **Console Application**). Em **Name**, coloque o nome do novo projeto e clique **OK**, como visto na figura:

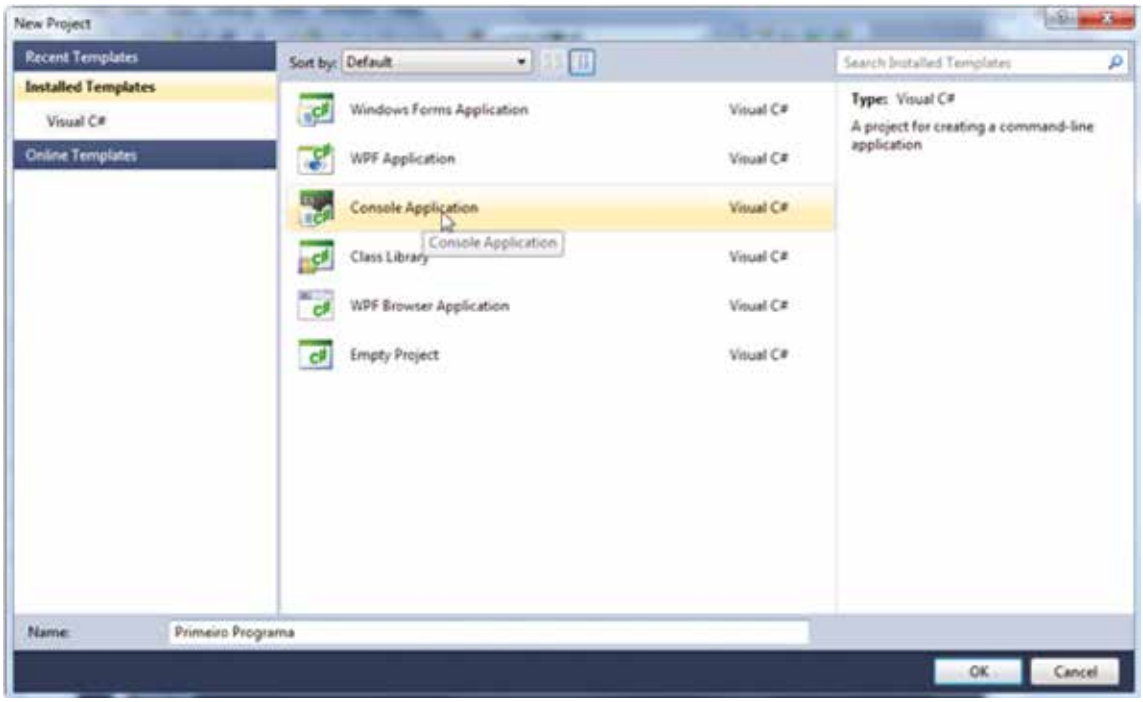


Figura 11 – Escolhendo o tipo de projeto e o seu nome

O C# está pronto para ser utilizado. Note que o *namespace* é o nome que foi dado no passo anterior. Nos programas procedurais, os comandos devem ser digitados no bloco do ***static void Main(Strings[] args)***:

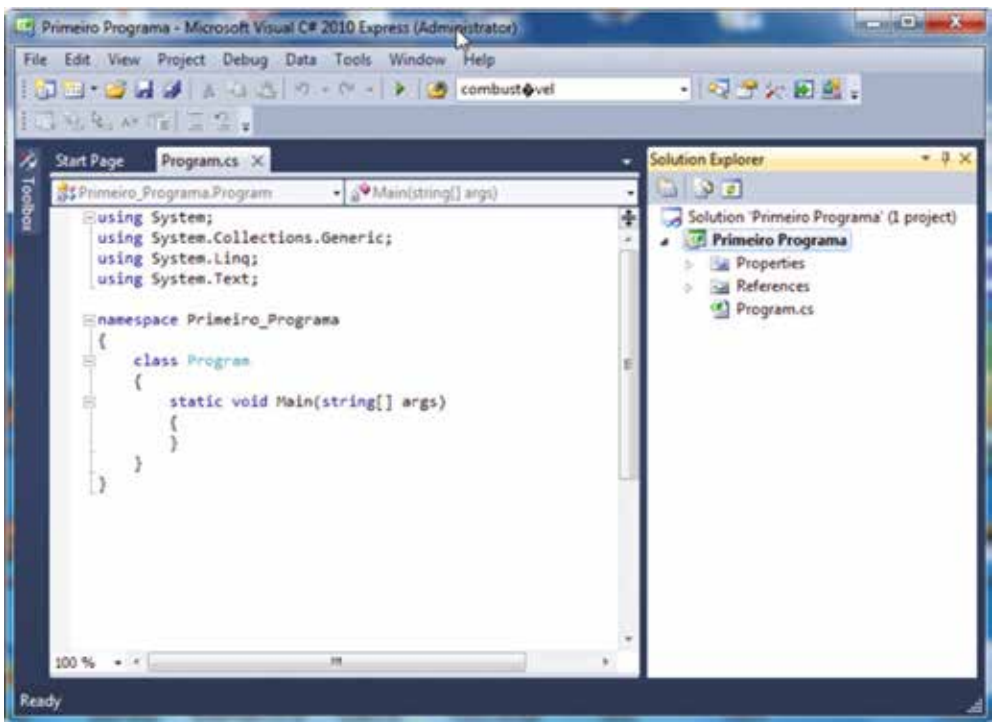


Figura 12 – Tela de edição do Visual C# 2010

O C# faz parte também de todos os pacotes do Visual Studio, sendo o Microsoft Visual Studio 2013 a última versão lançada até a elaboração deste livro-texto.

O processo de criação de um novo projeto é semelhante ao da versão Express. O passo 1, File/New, é igual. Veja:

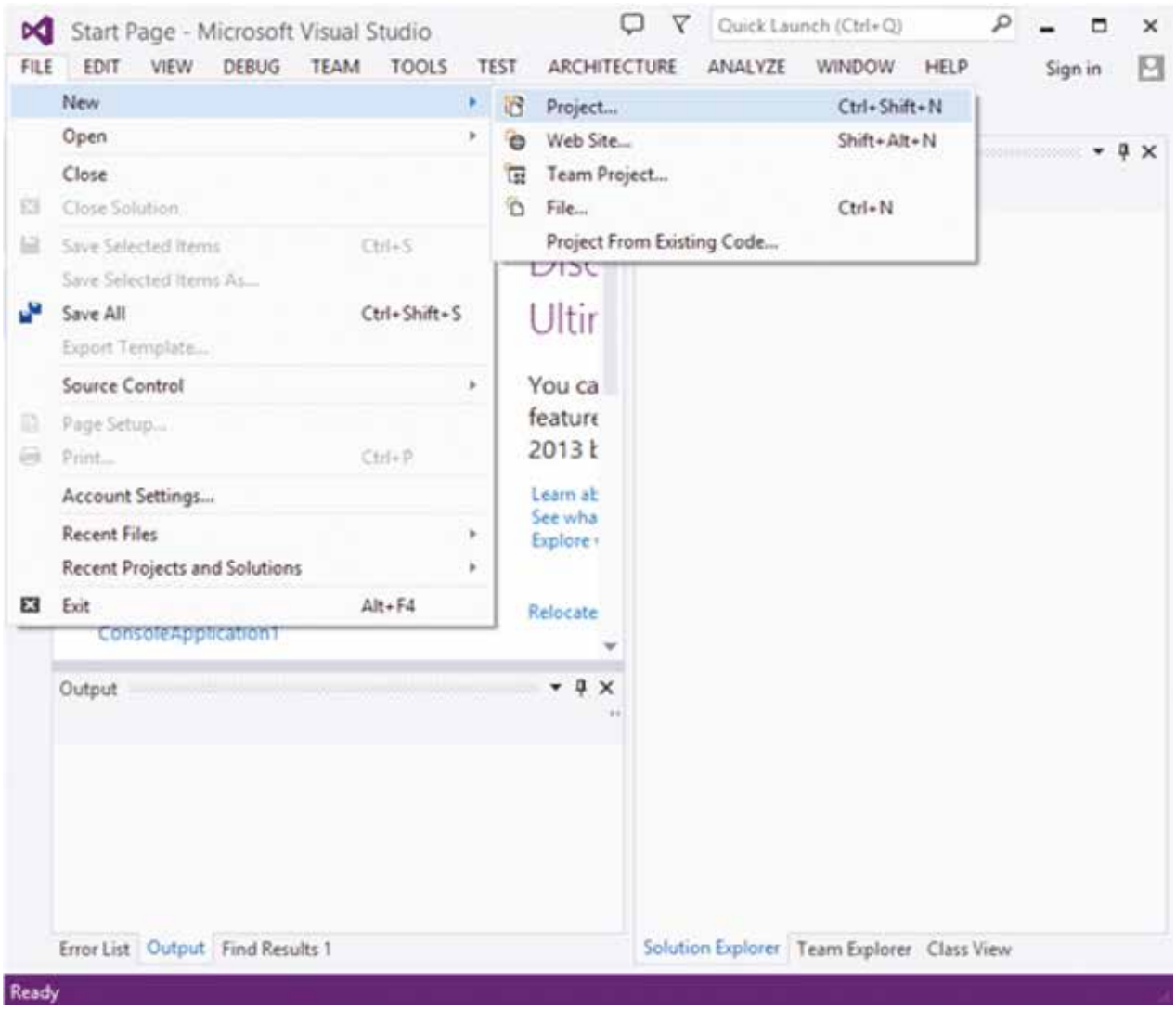


Figura 13 – Inicialização de um novo projeto no Visual Studio 2013

O passo seguinte requer um cuidado maior, pois o Visual Studio incorpora várias linguagens de programação. Em **Templates**, escolher **C#** e então proceder como na versão Express.

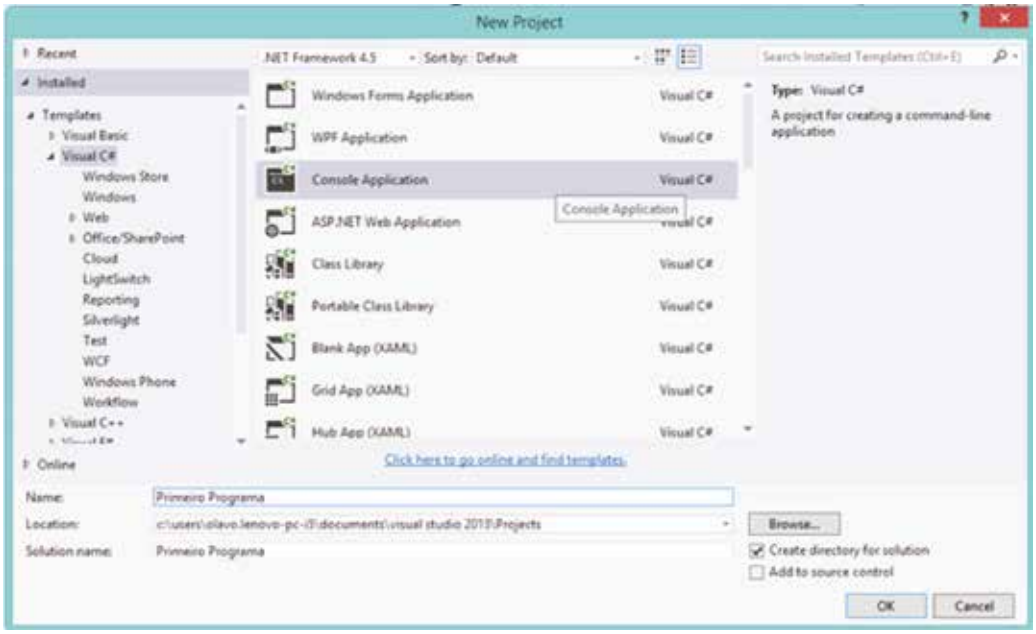


Figura 14 – Escolha do modelo de projeto

### 3.2.2 Execução C#

Como visto no passo 4 do item anterior, a edição do programa acontece no bloco do *Main*, mas muitas vezes a tela de edição perde a configuração devido à combinação de teclas que pode acontecer acidentalmente fechando alguma janela interessante. Nesses casos, tanto na versão Express quanto no Visual Studio, basta digitar em Windows/Reset Window Layout.

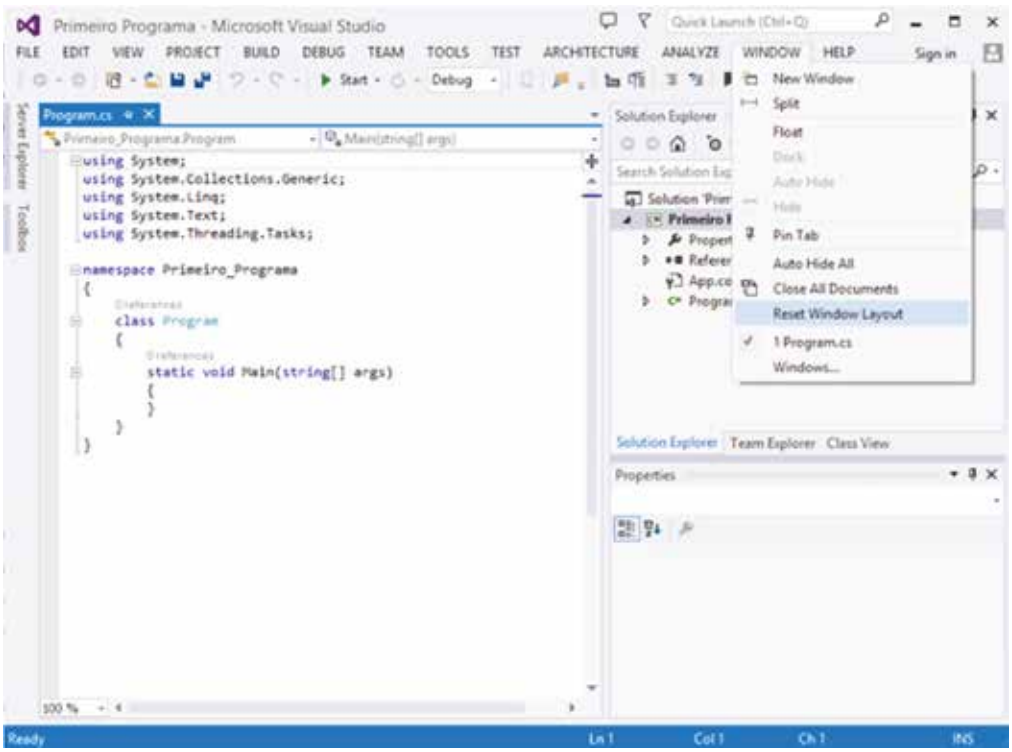


Figura 15 – Reiniciar as janelas

Para executar o programa digitado, basta ir em **Debug/Start Without Debugging** (Ctrl-F5) conforme a figura a seguir. Esse procedimento é adotado para evitar que a janela de saída se feche logo após a execução do programa, quando apenas o Start (F5) for escolhido.

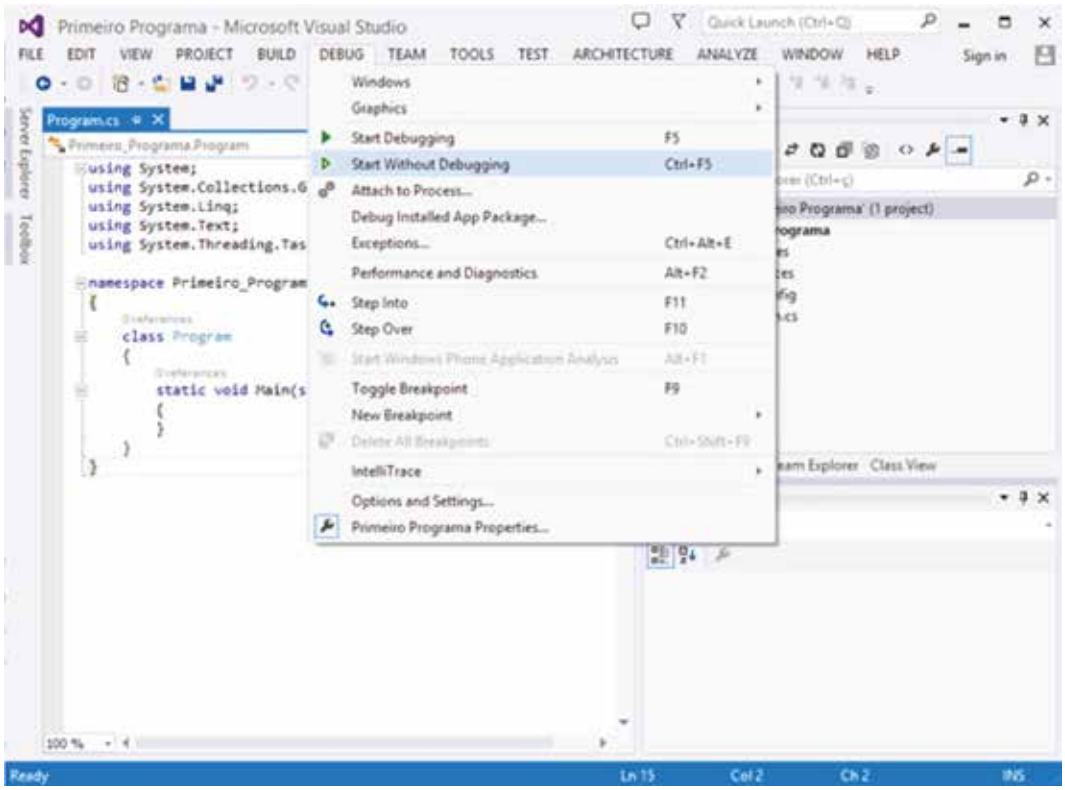


Figura 16 – Executando um programa

## 4 OS FUNDAMENTOS DA PROGRAMAÇÃO EM LINGUAGEM C#

A linguagem C# (pronuncia-se "C Sharp") faz parte desse conjunto de ferramentas oferecidas na plataforma .NET e surge como uma linguagem simples, robusta, orientada a objetos, fortemente tipada e altamente escalável, a fim de permitir que uma mesma aplicação possa ser executada em diversos dispositivos de *hardware*, independentemente de eles serem PCs, *tablets* ou qualquer outro dispositivo móvel. Além do mais, de acordo com Lima e Reis (2002), a linguagem C# também tem como objetivo permitir o desenvolvimento de qualquer tipo de aplicação: *web services*, Windows convencional, aplicações para serem executadas num *tablet* ou *smartphone* e também para internet, entre outros.

A proposta do C# em termos de linguagens de programação poderia ser descrita esboçando algumas das suas características principais:

- **Clareza, simplicidade e facilidade**: C# é clara, simples, fácil de aprender, mas nem por isso menos poderosa.

- É **completamente orientada a objetos**: C#, diferentemente de muitas linguagens existentes no mercado, é completamente orientada a objetos. Em C#, tudo é um objeto.

- **Não requer ponteiros para gerenciar a memória**: C# não requer ponteiros para alocar/desalocar memória *heap*. Esse gerenciamento, como dissemos anteriormente, é feito pelo GC (*Garbage Collector*).

- **Suporta interfaces, sobrecarga, herança, polimorfismo, atributos, propriedades e coleções, dentre outras características essenciais numa linguagem que se diz orientada a objetos**.

- **Possui código 100% reutilizável**: todo programa desenvolvido em C# é passível de reutilização a partir de qualquer outra linguagem de programação (LIMA; REIS, 2002).

A seguir, temos um trecho simples de código em C#:

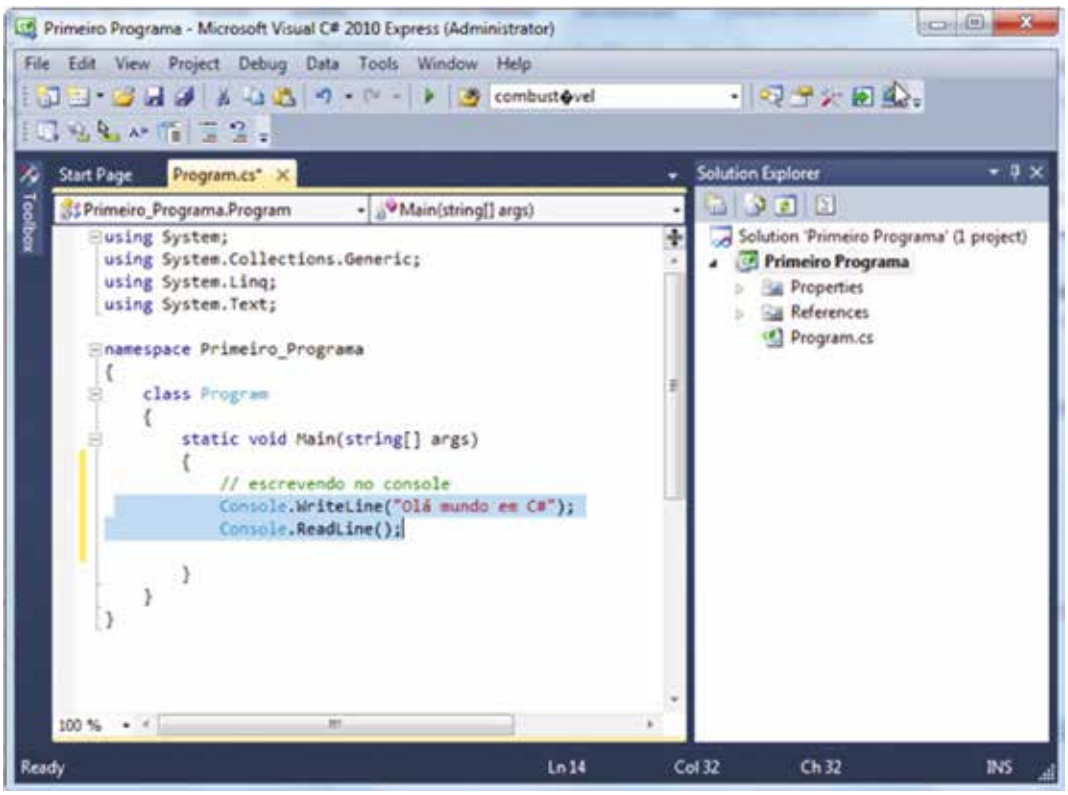


Figura 17 – Edição do primeiro código

O programa da figura anterior escreve a frase "Olá mundo em C#" em uma janela de console (*prompt* de comando). Ao executarmos, temos a tela vista na figura a seguir. Vamos estudá-la.

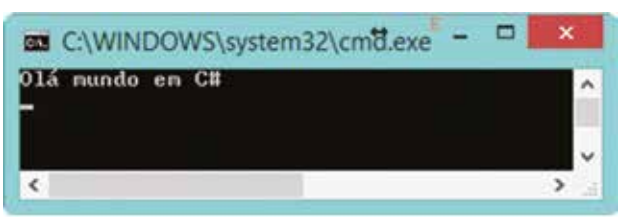


Figura 18 – Tela de saída do primeiro programa

Em sua primeira linha, temos a cláusula (ou palavra reservada ou comando) *using*. Sua função aqui é carregar os *namespaces* (classes ou bibliotecas) necessários para a execução do programa.

Após a definição das bibliotecas que serão utilizadas, iniciaremos a estrutura do programa. O programa principal também é considerado uma classe; deste modo, então utilizamos a cláusula *class* seguida pelo nome da classe, que, neste caso, será o nome do programa.

As chaves servem para definir o escopo de cada bloco de codificação, ou seja, determinam o início e o fim de cada sequência de comandos.

Por se tratar do programa principal, essa classe conterá apenas um método (função): *Main*(). Como toda linguagem da família C, "*main*" indica que o programa, ao ser executado, será iniciado a partir dele. É importante entender que a estrutura do programa principal será muito parecida com ao deste exemplo!

A cláusula *static* existe ali indicar ao .NET que essa função estará disponível permanentemente enquanto o programa estiver sendo executado. Já a cláusula *void* indica que o método *Main* não possui nenhum retorno.

**Observação**

Quando dizemos que a linguagem C# é fortemente tipada, é porque os nomes de variáveis, métodos, atributos e classes escritas com letras maiúsculas serão diferentes dos de letras minúsculas. Deste modo, o método *Main* deve ser escrito com a letra 'm' maiúscula, sempre.

O método *Main*() escrito conforme o exemplo significa também que o programa não receberá nenhum argumento (ou parâmetros) para sua execução. Caso seja necessário que o programa receba esse tipo de informação, a declaração desse método sofre uma ligeira alteração:

```
static void Main(string[] args)
```

Por fim, devemos nos atentar ao ponto-e-vírgula (;). Esse caractere deve ser colocado no final de cada linha de comando, indicando assim onde o comando é finalizado.

Se tratando de organização de código, temos os comentários, que podem ser escritos em linhas ou em blocos. Para comentários em linhas, utilizamos duas barras (//). Para comentários em bloco, usamos a sequência /* */, conforme o exemplo:

**Código 1 – Primeiro programa**

```
static void Main(string[] args)
{
    /*
    escrevendo no console
    Exemplo de declaração de método Main com
    Parâmetro: Main(string[] args)
    Não se esqueça de colocar um ; no final de cada
    linha de comando!
    */
     Console.WriteLine("Olá mundo em C#");
     Console.ReadLine();

}
```

Os comentários são observações que servirão unicamente para que o desenvolvedor faça anotações sobre seu trabalho ou sobre o código em si. Tudo que está dentro do comentário (seja em bloco ou em linha) será ignorado pelo compilador (incluindo comandos).

Para fins didáticos, utilizaremos o console para que possamos exemplificar os códigos e testarmos os conceitos. Isso torna o aprendizado mais simples e focado no que realmente é necessário.

Vejamos agora outro exemplo de código:

**Código 2 – Programa MeuTeste2, código com os comentários**

```
static void Main(string[] args)
{
    char c;
    string str;
    // Escreve no console sem retorno de carro
    Console.Write("Digite seu nome: ");
    //Lê uma string do console.<Enter> para concluir
    str = Console.ReadLine();
    // Escreve no console sem retorno de carro
    Console.Write("Digite uma letra:");
    // Lê do console um caractere simples.
    c = (char)Console.Read();
    // Escreve uma linha em branco
    Console.WriteLine();
    // Escreve uma string no console
    Console.WriteLine("Seu nome é: {0}", str);
    // Escreve 1 caractere com ToString() para converter
    Console.WriteLine("A letra é: {0}", c.ToString());
    Console.ReadLine();
}
```

Neste exemplo, encontramos uma rica quantidade de conceitos utilizados para programar em C#. Diferente do exemplo anterior, este programa agora escreve na tela, solicita a entrada de informações e devolve essas informações na tela.

Para isso, utilizamos a classe Console, que possui os métodos necessários para exibição de informações na tela e captura de dados via teclado, conforme ilustrado no quadro a seguir:

**Quadro 3 – Métodos de interação de teclado e tela**

| Método | Descrição |
|---|---|
| **Console.Write** | Escreve na tela, na posição onde está o cursor. |
| **Console.WriteLine** | Escreve na tela e ao final salta uma linha, posicionando o cursor no início da nova linha. |
| **Console.Read** | Lê a digitação de algo no teclado (aguarda até que uma tecla seja pressionada). |
| **Console.ReadLine** | Lê a digitação do teclado e aguarda até que a tecla Enter seja digitada. |

Outro conceito mostrado no código anterior é o das variáveis, que terão o escopo de onde elas foram definidas. No exemplo MeuTeste2, as duas variáveis declaradas (c e str) apenas funcionam dentro do método *Main*.

A tabela a seguir mostra os tipos de variáveis utilizados em C#.

**Tabela 1 – Tipos de variáveis em C#**

| Tipo | Valore possíveis |
|---|---|
| ***bool*** | Verdadeiro ou falso (valores booleanos) |
| ***byte*** | 0 a 255 (8 *bits*) |
| ***sbyte*** | -128 a 127 (8 *bits*) |
| ***char*** | Um caractere (16 *bits*) |
| ***decimal*** | ±1.0 x 10-28 a ± 7.9 x 1028 (128 *bits*) |
| ***double*** | ±5.0 x 10-324 a ± 1.7 x 10308 (64 *bits*) |
| ***float*** | ±1.5 x 10-45 a ± 3.4 x 1038 (32 *bits*) |
| ***int*** | -2.147.483.648 a 2.147.483.647 (32 *bits*) |
| ***uint*** | 0 a 4.294.967.295 (32 *bits*) |
| ***long*** | 9.223.373.036.854.775.808 a 9.223.372.036.854.775.807 (64 *bits*) |
| ***ulong*** | 0 a 18.446.744.073.709.551.615 (64 *bits*) |
| ***object*** | Qualquer tipo |
| ***short*** | -32.768 a 32.767 (16 *bits*) |
| ***ushort*** | 0 a 65.535 (16 *bits*) |
| ***string*** | Sequência de caracteres (16 *bits* por caractere) |

**Lembrete**

Para maiores detalhes sobre os dados desse quadro, verifique o Help da linguagem C#, que é encontrado no menu de opções do programa (último item ou digitando F1).

Para se trabalhar com variáveis, devemos tomar alguns cuidados e seguir algumas regras:

- O primeiro caractere deve ser uma **letra**.
- Se houver mais de um caractere, só podemos usar **letra** ou **algarismo.**
- Nomes de variáveis escritas com letras maiúsculas serão diferentes de letras minúsculas. Lembre-se: media é **diferente** de MEDIA.
- Nenhuma palavra reservada (nomes de comandos ou cláusulas) poderá ser nome de variável.

Exemplos:

```
int x;
bool opcao;
string texto1;
```

Apenas após a definição da variável é que podemos utilizá-las em nosso programa. Para atribuirmos valores para as nossas variáveis, devemos utilizar o símbolo de igualdade (=).

Exemplos:

```
x = 10; //Lê se: a variável x recebe o valor 10, ou, x recebe 10
opcao = false;
texto1 = "Boa noite";
```

Agora que a estrutura básica de um programa em C# foi apresentada, vejamos o próximo código. Este exemplo mostra a idade atual de uma pessoa e a que ela terá no ano que vem:

**Código 3 – Idade_v1**

```
static void Main(string[] args)
{
    int idade;
    idade = 15;

    // escrevendo no console
    Console.WriteLine("Programa Idade versão 1.0");
    Console.WriteLine();
    Console.WriteLine("A idade atual é: {0}", idade);
    Console.WriteLine();
    idade = idade + 1;
    Console.WriteLine("A idade ano que vem é: {0}", idade);
}
```

O programa Idade_v1, embora simples, ilustra mais alguns importantes fundamentos. Esse programa é iniciado declarando uma variável chamada idade do tipo inteiro (`int idade;`) e logo em seguida essa variável é iniciada (ou populada, de popular) com o valor 15 (`idade = 15;`), presumindo que a idade da pessoa é 15 anos.

Como o objetivo do programa é calcular a idade da pessoa para o ano que vem, é realizada uma operação matemática de adição (`idade = idade + 1;`), na qual é adicionado um ao valor já existente na variável idade, ou seja, a idade para o ano que vem é 16.

Em C#, é possível realizar operações matemáticas, conforme o quadro a seguir.

**Quadro 4 – Operadores matemáticos em C#**

| Símbolo | Operador matemático |
|---|---|
| + | Adição |
| - | Subtração |
| * | Multiplicação |
| / | Divisão |
| % | Resto da divisão |

**Lembrete**

Para maiores detalhes sobre os dados desse quadro e outras funções matemáticas, verifique o *Help* da linguagem C# no menu de opções do seu programa Visual C#.

Os programas MeuTeste2 e Idade_v1 também introduzem o conceito de máscara de exibição de valores ({0}). Essa máscara permite formatar melhor as mensagens de saída, deixando as mensagens mais simpáticas ao usuário do programa.

O conceito do uso desta máscara é bem simples. O conteúdo das chaves ({0}) é substituído pelo o conteúdo da variável passada como parâmetro, ou seja:

```
Console.WriteLine("A idade ano que vem é: {0}", idade);
```

Ela vai ser exibida para o usuário da seguinte maneira:

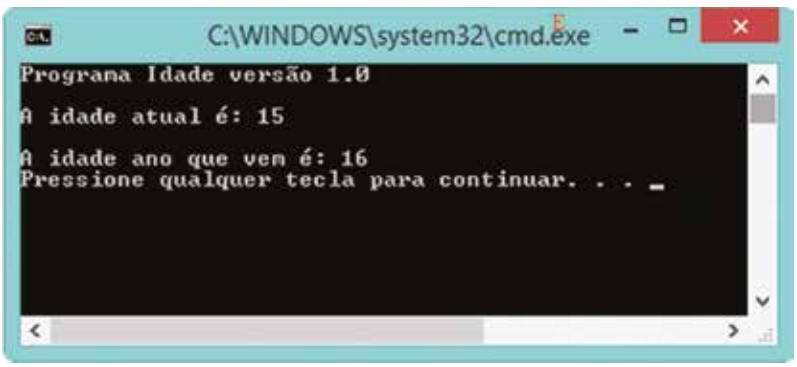


Figura 19 – Saída do programa V1.00

Podem ser montadas várias máscaras em uma mesma mensagem de exibição, conforme podemos ver nessa versão melhorada do programa Idade_v101 (versão 1.01):

**Código 4 – Idade_V1.01**

```
static void Main(string[] args)
{
    // declara a idade
    int idade, idadeanoquevem;
    idade = 15;
    idadeanoquevem = idade + 1;
    // escrevendo no console
    Console.WriteLine("Programa Idade versão 1.01");
    Console.WriteLine();
    Console.WriteLine("A idade atual é {0} e ano que vem será {1}",
        idade, idadeanoquevem);
    Console.WriteLine();
}
```

Para a versão nova, o programa Idade_v101 teve duas alterações. Para ilustrar o exemplo da máscara na mensagem de exibição com dois valores distintos, houve a necessidade de criar uma segunda variável

para armazenar a idade para o ano que vem. Deste modo, o valor resultante do cálculo da idade passou a ser armazenado na variável *idadeanoquevem* e o valor da idade atual, na variável *idade*. Essa divisão do cálculo em duas variáveis traz uma maior organização ao código do programa e permite utilizar uma mensagem única de saída com as suas respostas. Assim, a mensagem é definida como:

```
Console.WriteLine("A idade atual é {0} e ano que vem será {1}",
    idade, idadeanoquevem);
```

Essa técnica de se colocar máscara na mensagem se chama **formatação composta**. Uma cadeia de caracteres de formato composto consiste de sequências de texto fixo intercaladas com itens de formato. Tal recurso retorna uma nova cadeia de caracteres e o resultado é que cada item de formato é substituído pelo valor da variável correspondente na lista após a máscara. Assim, o item de formato {0} será substituído pelo primeiro valor após a máscara, ou seja, pelo conteúdo da variável idade, e o item {1} será substituído pelo do segundo valor depois da máscara, o conteúdo da variável idadeanoquevem.

**Saiba mais**

Para saber mais sobre a formatação composta, visite o *site* do Microsoft Developer Network no verbete "formatação composta". Lá poderão ser encontradas referências para os componentes de índice, de alinhamento e caracteres de formato. O *link* é:

FORMATAÇÃO composta. [s.d.]. Disponível em: <msdn.microsoft.com/pt-br/library/txafckwd(v=vs.110).aspx>. Acesso em: 27 out. 2014.

Assim, no programa Idade_v101 será exibida a tela a seguir:

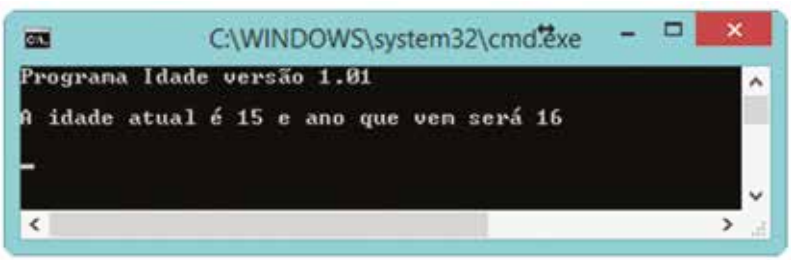


Figura 20 – Saída do programa Versão 1.01

Para a leitura de um valor via teclado, emprega-se a instrução Console.ReadLine();. Uma forma direta de aplicação é a atribuição do resultado dessa instrução à variável que receberá tal conteúdo.

```
nome = Console.ReadLine();
```

Dessa forma, o valor digitado pelo usuário será atribuído à variável "nome". No entanto, tal forma de aplicação é viável e fácil de ser utilizada quando o conteúdo da variável é um *string*, pois sua aplicação sempre retornará um valor desse tipo. Para os conteúdos de variáveis de outro tipo, faz-se necessária sua conversão para o tipo em que a variável foi declarada, antes ainda de sua atribuição.

```
float x = Convert.ToSingle(Console.ReadLine());
```

Observe que é feita a leitura do conteúdo da variável por meio do argumento da instrução para conversão do tipo *string* para *float* (ponto flutuante).

A forma de aplicação dessa instrução, sob o aspecto prático, será mais bem-ilustrada no próximo programa. O quadro a seguir mostra algumas das formas de conversão de *string* para outro tipo de variável.

**Quadro 5 – Conversões de *string* para tipos numéricos**

| Tipo numérico | Método | Método Parse |
|---|---|---|
| ***decimal*** | Convert.ToDecimal(*String*) | Decimal.Parse |
| ***float*** | Convert.ToSingle(*String*) | Single.Parse() |
| ***double*** | Convert.ToDouble(*String*) | Double.Parse() |
| ***short*** | Convert.ToInt16(*String*) | Int16.Parse() |
| ***int*** | Convert.ToInt32(*String*) | Int32.Parse() |
| ***long*** | Convert.ToInt64(*String*) | Int64.Parse() |
| ***ushort*** | Convert.ToUInt16(*String*) | Uint16.Parse() |
| ***uint*** | Convert.ToUInt32(*String*) | Uint32.Parse() |
| ***ulong*** | Convert.ToUInt64(*String*) | Uint64.Parse() |

Existe outra importante maneira de fazer conversões de tipo, o modelador *cast*.

O *cast* é um método pelo qual um valor é convertido a partir de um tipo de dados para outro. Ele pode ser executado para conversões numéricas em que o tipo de destino é de menor precisão ou que tenha um intervalo de valores menor. Também é utilizado para a conversão de instância de classe base para classe derivada.

Muitas vezes o *cast*, por ser uma conversão explícita em que o compilador é informado sobre a mudança, pode resultar em erros devido à falta de compatibilidade entre os tipos. Para testar a compatibilidade antes de realmente executar um *cast*, o C # fornece dois operadores:

- O operador 'is' verifica a possibilidade de fazer o *cast*, além de ser utilizado para determinar se um tipo de objeto é uma instância de uma determinada classe.

- O operador 'as' é utilizado para se obter o valor de *cast* sem efetivar a mudança de tipo. Dessa forma, é possível determinar se a operação pode ser efetuada com sucesso.

Voltando ao programa da idade, vamos agora realizar mais algumas implementações, de modo que ele fique mais prático, pois, embora para fins didáticos ele ilustre alguns conceitos importantes, o programa não possui uma utilidade muito prática. Vejamos agora o programa Idade_v2 (versão 2.0):

**Código 5 – Idade_V2.0**

```
static void Main(string[] args)
{
    // declara a idade
    int idade, idadenofuturo, qtdeanos;
    idade = 0;
    qtdeanos = 0;
    // escrevendo no console
    Console.WriteLine("Programa Idade versão 2.0");
    Console.WriteLine();
    Console.Write("Informe sua idade atual: ");
    idade = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine();
    Console.Write("Daqui a quantos anos você quer saber qual será sua idade? ");
    qtdeanos = Convert.ToInt32(Console.ReadLine());
    idadenofuturo = idade + qtdeanos;
    Console.WriteLine("Hoje você tem {0} anos. Daqui a {1} anos você terá {2} anos de idade!",
        idade, qtdeanos, idadenofuturo);
}
```

Agora o programa Idade_v2 possui um senso prático, pois, diferente do que acontecia nas versões anteriores, o usuário pode informar ao programa sua idade atual e solicitar para que seja calculada a sua idade dentro de determinada quantidade de anos (também informada pelo usuário). Logo, não é mais feito um cálculo com valores predeterminados dentro do programa.

A saída do programa deverá ocorrer como ilustra a figura a seguir:


Figura 21 – Saída da versão 2.0

**Observação**

É importante perceber que, assim como o aprendizado, o programa deve ser construído por etapas. A função básica do programa de cálculo de idade é a mesma nas três versões (Idade_v1, Idade_v101 e Idade_v2), porém o requinte na construção do código e as funções agregadas a cada versão foram construídos por etapas.

Estudando o código do programa Idade_v2, devemos nos atentar às variáveis. Embora seus valores sejam informados pelo usuário, todas as três variáveis são iniciadas logo após sua declaração.

Faz parte das boas práticas de programação iniciar as variáveis com valores predeterminados para garantir os valores iniciais das variáveis dentro do programa. Quando declaramos uma variável e esta não é inicializada pelo programador, as variáveis primitivas recebem um valor *default*. Outros tipos de variáveis, quando declaradas ou instanciadas, recebem como valor *default* o ***null***. Veja:

**Quadro 6 – Padrões de valores de tipos de variáveis em C#**

| Tipo | Conteúdo |
|---|---|
| *Boolean* | **False** |
| *Byte* | **0** |
| *Short* | **0** |
| *Char* | ***Null*** |
| *Int* | **0** |
| *Float* | **0.0** |
| *Long* | **0** |
| *Double* | **0.0** |

Agora sabemos como é a estrutura básica de um programa em C#, como se comportam as variáveis e como fazer o programa interagir com o usuário. O próximo exemplo ilustrará como trabalhar com textos (*string*):

**Código 6 – Programatexto**

```
static void Main(string[] args)
{
    string c, c1, d, d1;
    Console.Write("\n Digite a palavra 1: ");
    c = Console.ReadLine();
    Console.Write("\n Digite a palavra 2: ");
    c1 = Console.ReadLine();
    Console.Write("\n tamanho da 1 palavra: {0}", c.Length);
    Console.Write("\n concatenando sem armazenar: {0}", c + c1);
    d = c;
    Console.Write("\n o conteúdo de d: {0}", d);
    d1 = c + c1;
    Console.Write("\n concatenação: {0}", d1);
    Console.Write("\n primeiro caractere: {0}", c.Substring(0, 1));
    Console.Write("\n último caractere: {0}", c1.Substring(c1.Length - 1, 1));
    Console.Write("\n todos menos o primeiro: {0}", d1.Substring(1));
    Console.Write("\n o terceiro elemento: {0}", c.Substring(4, 1));// 5º caracter
    Console.Write("\n os três primeiros elementos: {0}", c.Substring(0, 3));
    Console.Write("\n os três últimos elementos: {0}", c1.Substring(c1.Length - 3, 3));

}
```

O programa Programatexto ilustra alguns exemplos de como trabalhar com dados do tipo texto (*string*), verificando o tamanho do texto digitado, concatenando textos diferentes e trabalhando com partes dos textos, conforme mostra o quadro a seguir.

**Quadro 7 – Métodos de manipulação de dados do tipo texto (*string*)**

| Função | Descrição |
|---|---|
| **Substring(posição_inicial, tamanho)** | Este método retorna um pedaço do texto (*substring*) contido na variável (ou atributo) do tipo *string*. A *substring* começa em uma posição de caractere especificada (`posição_inicial`) e tem um comprimento especificado (`tamanho`), lembrando que a primeira posição da *string* (primeiro caractere do texto) é indicada como posição 0. |
| **Length** | Esta propriedade (atributo) contém o número de caracteres – ou seja, o tamanho – do texto (*string*) atual. |

**Lembrete**

Essa lista não está completa; por isso, verifique o *Help* da linguagem C# no menu de opções do seu programa Visual C# para obter maiores detalhes e conhecer outras funções.

Algumas observações importantes devem ser feitas em relação ao tipo de dado *string*. Em C#, *strings* são objetos do *tipoSystem.String* cujo respectivo *alias* é *string*. A classe *System.String* define métodos para manipular *strings* como concatenação, tamanho da *string*, cópia etc. Em C#, *strings* não podem ser modificadas; uma vez criadas e inicializadas, qualquer modificação efetuada não é feita diretamente na *string*. Entretanto, o método usado na operação retorna uma nova *string* com as modificações à *string* original. Isso equivale a dizer que operações com *string* são custosas porque além de estarmos lidando com objetos, tipos referência, cada operação que modifique a *string* vai retornar um novo objeto (LIMA; REIS, 2002).

A saída para as palavras "Universidade Paulista" inseridas no programa é apresentada na figura a seguir:


Figura 22 – Saída do Programatexto

A partir de agora, veremos os aspectos mais "inteligentes" da programação, ou seja, falaremos sobre desvio condicional e laços.

O desvio condicional ou tomada de decisão, como em qualquer outra linguagem de programação, é a característica responsável por verificar condições dentro do programa. Vejamos o exemplo a seguir:

**Código 7 – Exemplo desvio condicional**

```
static void Main(string[] args)
{
    double numero = 0;
    Console.Write("Digite um número: ");
    numero = double.Parse(Console.ReadLine());
    if (numero > 20)
    {
        Console.WriteLine("Metade: {0}", numero / 2);
    }
}
```

O exemplo Testedecondicao ilustra uma maneira muito simples de utilização de um desvio condicional. Após a digitação de um número informado pelo usuário (variável `numero`), o programa verifica se o número digitado é maior que 20 (`if (numero > 20)`). Sendo verdade essa condição, o programa exibe o valor referente a metade do número digitado. Se o número for igual ou menor a 20, o programa não executa nenhum procedimento.

Portanto, um desvio condicional em C# possui a seguinte estrutura:

```
if (condição)
{
   < seqüência de comandos separados por ; >
}
else
{
   < seqüência de comandos separados por ; >
}
```

Vejamos agora outro exemplo de desvio condicional:

**Código 8 – Testedecondição2**

```
static void Main(string[] args)
{
    double numero = 0;
    Console.Write("Digite um número: ");
    numero = Convert.ToDouble(Console.ReadLine());
    if (numero % 2 == 0)
    {
        Console.WriteLine("PAR");
    }
    else
    {
        Console.WriteLine("IMPAR");
    }
}
```

No exemplo Testedecondicao2, o usuário digita um número e o programa determina, por meio de um desvio condicional, se o número informado é par ou ímpar. Então lê-se o desvio condicional como: "Se o resto da divisão do número informado for igual a zero, faça <condição 1>; se não, faça a <condição 2>", sendo que a <condição 1> é o primeiro bloco de comandos (condição verdadeira) e a <condição 2> refere-se ao segundo bloco de comandos (condição falsa).

Assim, ao executarmos o programa, conforme o valor de entrada, é mostrado um dos valores (veja as figuras a seguir).

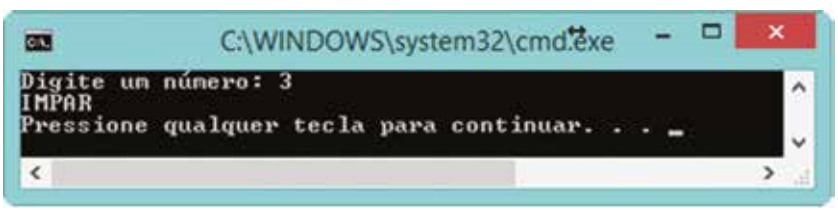


Figura 23 – Saída quando um número ímpar é digitado

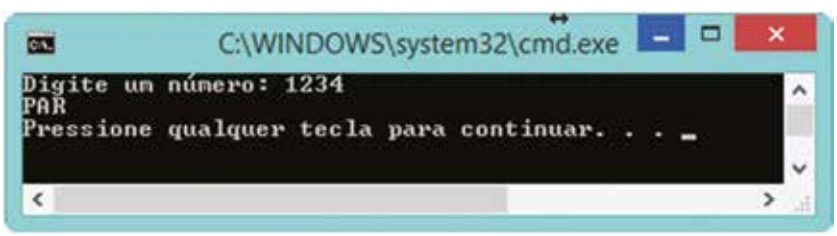


Figura 24 – Saída quando um número par é digitado

O desvio condicional pode assumir ainda uma terceira estrutura:

**Código 9 – Testedecondicao3**

```
static void Main(string[] args)
{
    int a, b = 0;
    Console.Write("Digite um número: ");
    a = Convert.ToInt32(Console.ReadLine());
    Console.Write("Digite outro número: ");
    b = Convert.ToInt32(Console.ReadLine());
    if (a > b)
    {
        // Bloco de comandos 1
        Console.WriteLine("O primeiro número é maior que o segundo número!");
    }
    else if (a < b)
    {
        // Bloco de comandos 2
        Console.WriteLine("O segundo número é maior que o primeiro número!");
    }
    else
    {
        // Bloco de comandos 3
        Console.WriteLine("Os dois números são iguais!");
    }
}
```

O exemplo Testedecondicao3 mostra a estrutura de desvio condicional com três blocos de comandos, sendo que:

- O bloco de comandos 1 refere-se à condição verdadeira do primeiro desvio condicional (`a > b`).
- O bloco de comandos 2 refere-se à condição falsa do primeiro desvio condicional (`a > b`) e à condição verdadeira do segundo desvio condicional (`a < b`).
- O bloco de comandos 3 refere-se à condição falsa do primeiro desvio condicional (`a > b`) e à condição falsa do segundo desvio condicional (`a < b`).

Ao executarmos, observamos a seguinte tela:

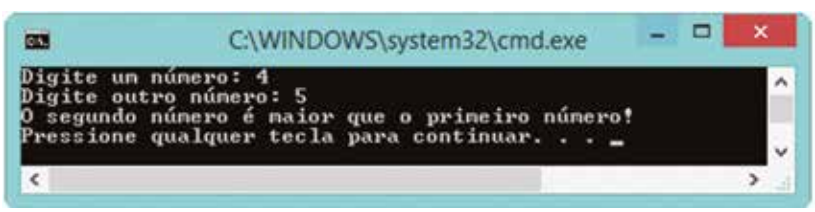


Figura 25 – Saída do programa testedecondição3

Podemos analisar o mesmo programa com outra estrutura de desvio condicional, tal como:

**Código 10 – testedecondicao4**

```
static void Main(string[] args)
{
    int a, b = 0;
    Console.Write("Digite um número: ");
    a = Convert.ToInt32(Console.ReadLine());
    Console.Write("Digite outro número: ");
    b = Convert.ToInt32(Console.ReadLine());
    if (a > b)
    {
        // Bloco de comandos 1
        Console.WriteLine("O primeiro número é maior que o segundo número!");
    }
    if (a < b)
    {
        // Bloco de comandos 2
        Console.WriteLine("O segundo número é maior que o primeiro número!");
    }
    if (a == b)
    {
        // Bloco de comandos 3
        Console.WriteLine("Os dois números são iguais!");
    }
}
```

O programa Testedecondicao4, em termos de objetivo e tratamento lógico, é igual ao programa Testedecondicao3. Porém, sua estrutura dos desvios condicionais é diferente! Este exemplo serve apenas para ilustrar como é possível construir diversas estruturas lógicas para o desvio condicional.

Ao testar o programa Testedecondição4, temos a saída na figura a seguir:

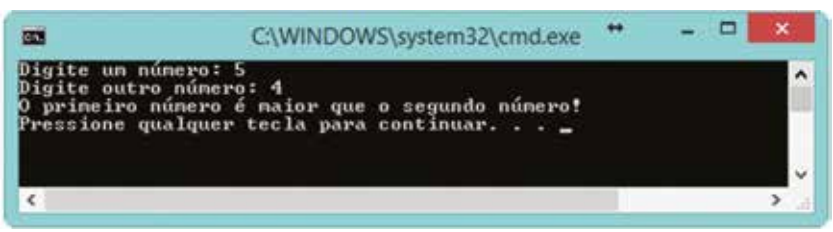


Figura 26 – Saída do programa Testedecondição4 usando 5 e 4 como entrada

**Observação**

O objetivo deste material não é a lógica de programação. Portanto, não discutiremos aqui qual a melhor forma de desenvolver os aspectos lógicos da estrutura dos programas, apenas apresentaremos as possíveis opções de como montar tais estruturas.

Quando falamos em desvio condicional, torna-se necessário falarmos também sobre operadores lógicos e relacionais, conforme é mostrado no quadro a seguir.

**Quadro 8 – Operadores lógicos e relacionais em C#**

| Operador | Tipo | Descrição |
|---|---|---|
| == | Relacional | Igual a. Comparação de igualdade. |
| != | Relacional | Diferente de. Comparação de desigualdade. |
| > | Relacional | Maior que. |
| < | Relacional | Menor que. |
| >= | Relacional | Maior ou igual a. |
| <= | Relacional | Menor ou igual a. |
| && | Lógico | Conjunção. Operador lógico e. É verdade se todas as condições forem verdadeiras. |
| \|\| | Lógico | Disjunção. Operador lógico ou. É verdadeira se pelo menos uma condição for verdadeira. |
| ! | Lógico | Negação. Operador lógico not. Inverte o valor da expressão ou condição; se verdadeira, inverte para falsa e vice-versa. |

Não é demais relembrar também o comportamento dos operadores lógicos E, OU e NOT, conforme ilustram os seguintes quadros:

**Quadro 9 – Tabela verdade para o operador lógico E (conjunção)**

| Condição 1 | Condição 2 | Resultado lógico |
|---|---|---|
| Falso | Falso | Falso |
| Falso | Verdadeiro | Falso |
| Verdadeiro | Falso | Falso |
| Verdadeiro | Verdadeiro | Verdadeiro |

**Quadro 10 – Tabela verdade para o operador lógico OU (disjunção)**

| Condição 1 | Condição 2 | Resultado lógico |
|---|---|---|
| Falso | Falso | Falso |
| Falso | Verdadeiro | Verdadeiro |
| Verdadeiro | Falso | Verdadeiro |
| Verdadeiro | Verdadeiro | Verdadeiro |

**Quadro 11 – Tabela verdade para o operador lógico NOT (negação) aplicado a condições de disjunção (OU)**

| Condição 1 | Condição 2 | Resultado lógico negado |
|---|---|---|
| Falso | Falso | Verdadeiro |
| Falso | Verdadeiro | Falso |
| Verdadeiro | Falso | Falso |
| Verdadeiro | Verdadeiro | Falso |

**Quadro 12 – Tabela verdade para o operador lógico NOT (negação) aplicado a condições de conjunção (E)**

| Condição 1 | Condição 2 | Resultado lógico negado |
|---|---|---|
| Falso | Falso | Verdadeiro |
| Falso | Verdadeiro | Verdadeiro |
| Verdadeiro | Falso | Verdadeiro |
| Verdadeiro | Verdadeiro | Falso |

Existe outro tipo de desvio condicional em C# chamado *switch* (ou *case*). Esse desvio condicional não se utiliza de expressões relacionais, ele simplesmente verifica se a condição possui um determinado valor para tomar a decisão. Necessariamente, esse valor deverá ser um valor do tipo numérico inteiro ou *string*.

A estrutura do desvio condicional do tipo *switch* segue o seguinte padrão:

```
switch (condição)
{
   case valor1:
      < seqüência de comandos separados por ; >
   break;
   case valor2:
      < seqüência de comandos separados por ; >
   break;
   case valorN:
      < seqüência de comandos separados por ; >
   break;
}
```

Dentro do bloco principal do *switch* pode haver quantos blocos *case* forem necessários para realizar as devidas comparações. Porém, alguns pontos devem ser observados:

- Em C#, o comando *break*, no final de cada sequência de comandos, de cada bloco *case*, deve ser inserido, indicando o final da sequência de comandos do respectivo bloco *case*.
- Pode haver um bloco *default* após todos os blocos *case*. Esse bloco será executado caso nenhuma condição *case* seja válida.

- Não pode haver mais de um bloco *case* verificando o mesmo valor da condição.

**Código 11 – Testeswitch**

```
static void Main(string[] args)
{
  {
   int a, b, c = 0;
   Console.WriteLine("Digite um número: ");
   a = Convert.ToInt32(Console.ReadLine());
   Console.WriteLine("Digite outro número: ");
   b = Convert.ToInt32(Console.ReadLine());
   Console.WriteLine("Agora, o que você quer fazer com esse números? ");
   Console.WriteLine("1 - Verificar se o primeiro número é maior que o segundo número.");
   Console.WriteLine("2 - Verificar se o segundo número é maior que o primeiro número.");
   Console.WriteLine("3 - Verificar se os números são iguais.");
   c = Int32.Parse(Console.ReadLine());
   switch (c)
   {
       case 1:
           if (a > b)
           {
               Console.WriteLine("O primeiro número é maior que o segundo número!");
           }
           else
           {
               Console.WriteLine("O primeiro número não é maior que o segundo número!");
           }
           break;
       case 2:
           if (a < b)
           {
               Console.WriteLine("O segundo número é maior que o primeiro número!");
           }
           else
           {
               Console.WriteLine("O segundo número não é maior que o primeiro número!");
           }
           break;
       case 3:
           if (a == b)
           {
               Console.WriteLine("Os dois números são iguais!");
           }
           else
           {
               Console.WriteLine("Os dois números não são iguais!");
           }
           break;
       default:
           Console.WriteLine("Opção inválida!");
           break;
   }
   Console.WriteLine("Pressione qualquer tecla para sair...");
  }
}
```

O programa Testeswitch mostra não só a aplicação do desvio condicional *switch*, mas também o uso integrado de outros desvios condicionais dentro de cada bloco *case*.

É possível verificar que o programa Testeswitch, em termos de escopo, faz a mesma coisa que o programa Testecondicao4, porém agora o usuário, além de informar os números a serem analisados, também informa qual o tipo de análise que deve ser feito com esses números. O programa ainda verifica, caso não for escolhida nenhuma das opções de análise disponíveis, a opção inválida, avisando o usuário sobre a escolha (utilizando o bloco *default*).

**Observação**

As tomadas de decisões fazem os programas assumirem o aspecto de inteligência. As bases da inteligência artificial estão justamente nos arranjos matemáticos e lógicos dos programas de computadores. Aprenda a trabalhar bem com as ferramentas de tomada de decisão para criar programas de maior eficácia na resolução dos problemas de seu escopo de atuação!

Outro conjunto de ferramentas importante para a programação são os laços, também chamados de loops. Em linguagem C#, temos os seguintes tipos de laços:

- *for*;
- *foreach*;
- *while*;
- *do while*.

O laço *for* possui três componentes para controle de suas iterações (voltas do laço), sendo que:

- O primeiro componente se refere à declaração da variável de controle do laço.
- O segundo componente é a condição de parada do laço. É a expressão lógica, com base na variável de controle do laço, que será testada durante a execução do laço (cada iteração).
- O terceiro componente é a manipulação da variável de controle do laço que determinará o passo de cada iteração.

Vejamos um exemplo:

**Código 12 – Testafor**

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine("Iteração: {0}", i);
    }
}
```

No exemplo Testafor, temos um laço *for* em que a variável de controle do laço (também chamada de variável contadora) foi declarada e inicia em zero. A condição de laço é dada por `i<5`, ou seja, no momento em que a variável de controle do laço chegar no valor 5, o laço não mais será executado. Por fim, a manipulação da variável contadora se dará por um autoincremento de uma unidade (`i++`) em relação ao valor anterior.

Deste modo, será apresentada ao usuário uma contagem de zero a quatro na tela:

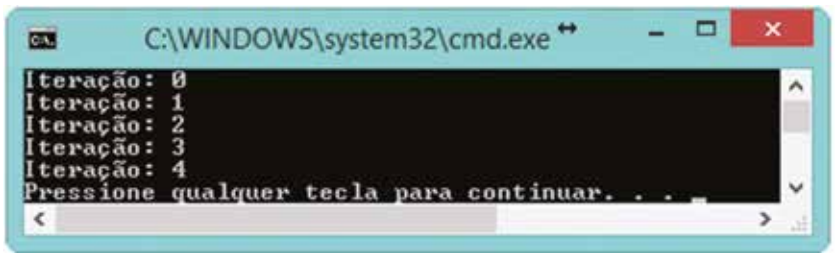


Figura 27 – Saída das iterações do laço *for*

O laço *for* permite alguns controles secundários em relação às suas iterações. Vejamos:

**Código 13 – Testafor2**

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        if (i == 3)
        {
            break;
        }
        Console.WriteLine("Iteração: {0}", i);
    }
}
```

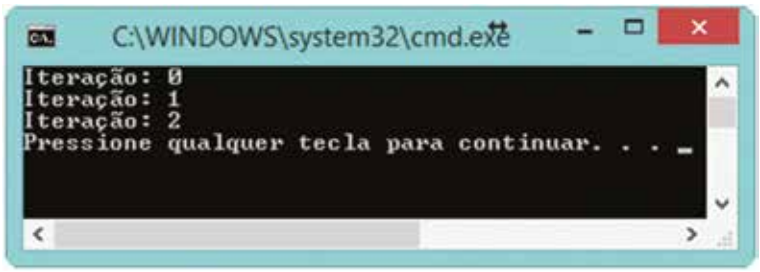


Figura 28 – Interrompendo as iterações

**Código 14 – Testafor3**

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        if (i == 3)
        {
            continue;
        }
        Console.WriteLine("Iteração: {0}", i);
    }
}
```

Os programas Testafor2 e Testafor3 apresentam o uso dos comandos *break* e *continue* dentro do laço *for*.

O comando *break* interrompe o laço e dá continuidade ao programa. No exemplo Testafor2, o laço será interrompido quando a variável contadora chegar a 3, conforme a figura anterior.

Já o comando *continue* interrompe a iteração corrente, não o laço todo. No exemplo Testafor3, quando a variável contadora chegar em 3, essa iteração não será executada, mas as próximas iterações serão executadas normalmente até o final do laço (`i<5`) (veja a figura a seguir).

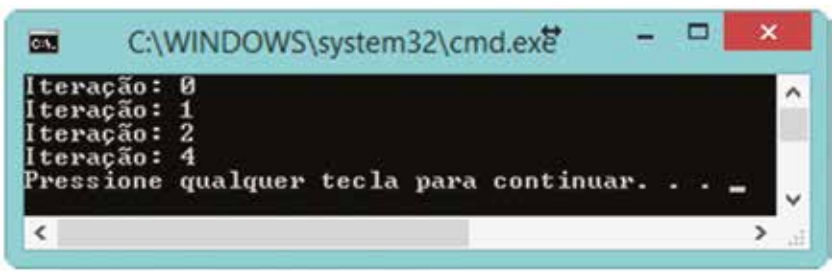


Figura 29 – Programa desviando para o fim do laço

**Lembrete**

O laço *for* pode retroceder, bastando inicializar a variável de controle com um número maior que o valor da comparação de parada e fazer a operação de passo utilizando as operações – ou -=.

Vejamos o código a seguir:

**Código 15 – Testafor4**

```
static void Main(string[] args)
{
    string[] Texto = new string[5];
    Texto[0] = "palavra 1";
    Texto[1] = "palavra 2";
    Texto[2] = "palavra 3";
    Texto[3] = "palavra 4";
    Texto[4] = "palavra 5";
    int i = 0;
    foreach (string s in Texto)
    {
        Console.WriteLine("Iteração: {0} - {1}", i, s);
        i++;
    }
}
```

O programa Testafor4 traz a aplicação de duas técnicas interessantes: a utilização de vetores (*arrays*) e de um laço específico para vetores, o *foreach*.

Em C#, os vetores são objetos e, como tal, são declarados e instanciados (veremos posteriormente mais detalhes sobre declaração e instanciamento de objetos). No caso, foi declarado um vetor chamado textos de cinco posições (`string[] textos = new string[5]`). Em cada posição do vetor (lembrando que a primeira posição inicia-se em zero), foi atribuído um valor do tipo *string* (um texto). Por fim, foi declarada uma variável que será utilizada como variável contadora para o próximo laço, o *foreach*.

O laço *foreach* é indicado em casos em que se faz necessário percorrer todos os elementos de um vetor, pois sua condição de parada do laço se dá quando são percorridas todas as posições do vetor. Para isso, o laço *foreach* solicita que seja passado como parâmetro uma variável de controle para a captura do valor de cada elemento do vetor, tratando assim os dados daquele elemento como uma simples variável (`string s in textos`). Perceba que a variável contadora (`i`) é utilizada meramente para fins didáticos para mostrar em que posição está o laço. Para o usuário será mostrado o seguinte resultado:

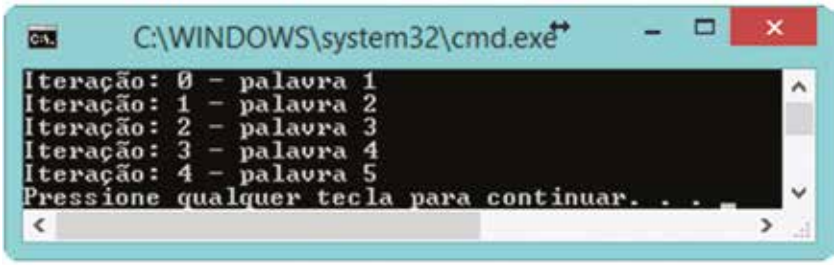


Figura 30 – Saída das iterações do comando *foreach*

O próximo laço que estudaremos é o *while*. O laço *while* é usado quando não sabemos o número de vezes que devemos executar um bloco de código, mas apenas a condição que deve ser satisfeita para executar o bloco dentro do *while*. Essa condição é uma expressão booleana que deverá ser verdadeira para garantir pelo menos a primeira ou a próxima iteração (LIMA; REIS, 2002).

**Código 16 – *Testawhile***

```
static void Main(string[] args)
{
    int i = 1;
    while (i < 10)
    {
        Console.WriteLine("Iteração: {0}", i);
        i++;
    }
}
```

O exemplo *Testawhile* mostra uma contagem de 1 a 99. A condição de parada do laço é `i < 10`, logo, a saída apresentada ao usuário será:

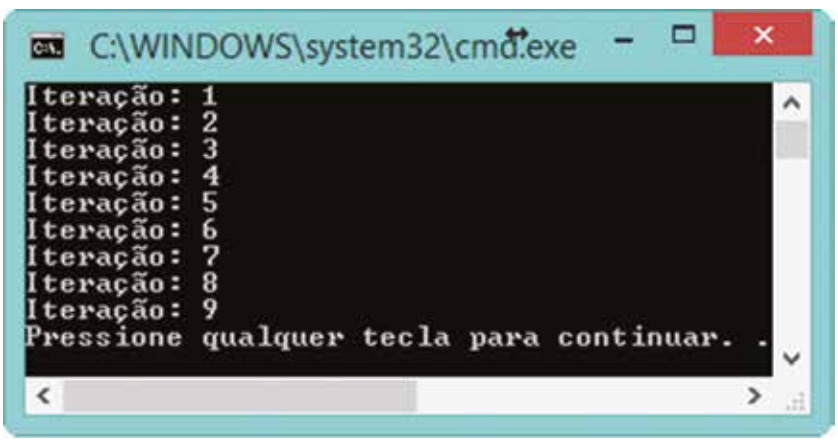


Figura 31 – Saída do programa *Testawhile*

Quando a variável `i` chegar ao valor de 100, a condição do laço será falsa, indicando assim que é o momento de terminar o laço e não executar mais o bloco de comandos do laço *while*.

O laço *while* permite o uso dos comandos *break* e *continue*, da mesma forma do laço *for* (já explicado anteriormente).

Por fim, o laço *do-while* é usado quando queremos que um bloco de código seja executado pelo menos uma vez. A condição a ser satisfeita se encontra no fim do bloco de código e não no começo, como no caso dos laços *for* e *while* (LIMA; REIS, 2002).

**Código 17 – *Testadowhile***

```
static void Main(string[] args)
{
    int i = 1;
    do
    {
        Console.WriteLine("Iteração: {0}", i);
        i++;
    } while (i < 10);
}
```

O programa *Testadowhile* realiza a mesma contagem que o programa *Testawhile*, porém o comportamento apresentado ao usuário será diferente. Como o laço *do-while* primeiro executa o bloco de comandos para depois verificar a condição do laço, a saída será apresentada ao usuário será:

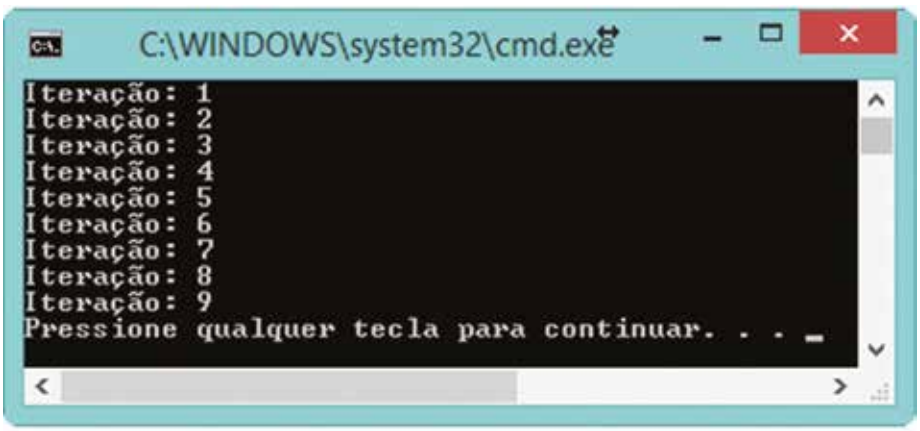


Figura 32 – Saída do programa *Testadowhile*

Em resumo, a diferença entre o laço *while* e o laço *do-while* é que o primeiro testa a condição e depois executa o bloco de comandos, enquanto o segundo executa o bloco de comandos e depois testa a condição.

Quando queremos guardar diversos objetos e informações, podemos fazer uso de matrizes. Uma matriz (*array*) é uma estrutura de dados que consegue guardar vários elementos e ainda nos possibilita pegar esses elementos de maneira fácil.

Veja a criação de uma matriz de 5 posições de números inteiros:

```
int[] numeros = new int[5];
```

Para guardar elementos nessas posições, fazemos:

```
numeros[0] = 1;
numeros[1] = 600;
numeros[2] = 257;
numeros[3] = 12;
numeros[4] = 42;
Console.WriteLine(numeros[1]);
```

A primeira posição de um *array* é 0 (zero). Logo, as posições de um *array* vão de 0 (zero) até (tamanho-1).

Podemos criar e logo em seguida inicializar o conteúdo de uma *array*, e o C# nos oferece um atalho para isso. Para preencher um *array* de inteiros preenchido com os números de 1 a 5, poderíamos utilizar o seguinte código:

```
int[] umAoCinco = new int[] { 1, 2, 3, 4, 5 };
```

A manipulação de um *array* não é fácil. Por exemplo, em um *array* com 5 elementos armazenados, se quisermos remover o valor armazenado na posição 1, precisaremos reordenar todo o nosso *array*. Para inserir um elemento no meio do *array*, precisamos "abrir um espaço" dentro da *array*, empurrando os elementos posteriores para os seguintes, para aí sim colocar o novo elemento no meio, isso se tivermos espaço livre dentro a *array*.

Para contornarmos as dificuldades de trabalhar com uma *array*, o C# oferece uma classe chamada *list* ou lista enumerada. Não nos preocupemos com a questão de ser uma classe, por enquanto será apenas uma técnica que pode ser utilizada sem o conhecimento de OO. Para utilizarmos uma lista dentro do código, precisamos informar qual é o tipo de elemento que a lista armazenará:

```
List<int> lista = new List<int>();
```

Da mesma forma que criamos a lista de inteiros, também poderíamos criar uma de qualquer outro tipo ou classes do C#. Essa lista do C# armazena seus elementos dentro de um *array*.

Agora que instanciamos o *list*, podemos utilizar o método *Add* para armazenar novos elementos:

```
// c1 fica na posição 0
lista.Add(1);
// c2 na 1
lista.Add(2);
// e c3 na 2
lista.Add(3);
```

Se acessar um elemento fazemos como num *array*, ou seja, pelo índice da sua posição (no caso, 0, igual no *array*):

```
int j = lista[0];
```

Se quisermos remover uma das contas da lista, podemos usar o método *Remove* ou *RemoveAt*:

```
lista.Remove(2); // remove pelo elemento

lista.RemoveAt(0); // remove pelo índice
```

Para saber quantos elementos existem em nosso *list*, podemos simplesmente ler a propriedade *count*:

```
int qtdDeElementos = lista.Count;
```

Também podemos descobrir se um elemento está dentro de uma lista:

```
bool tem1 = lista.Contains(1); // true
bool tem7 = lista.Contains(7); // false
```

Outro recurso que a classe *list* nos fornece é a iteração em cada um dos seus elementos pelo *foreach*:

```
foreach (int i in lista)
{
    Console.WriteLine(i);
}
```

Além de Add, Remove, RemoveAt e Contains, a lista enumerada possui outras funções, conforme visto no quadro a seguir:

**Quadro 13 – Principais funções da classe <List>**

| Nome | Descrição |
|---|---|
| **Add** | Adiciona um objeto ao final de List<T>. |
| **AddRange** | Adiciona os elementos da coleção especificada ao final de List<T>. |
| **Clear** | Remove todos os elementos de List<T>. |
| **Contains** | Determina se um elemento está em List<T>. |
| **CopyTo(T[])** | Copia List<T> inteiro em uma matriz unidimensional compatível, a partir do início da matriz de destino. |
| **Equals(Object)** | Verifica se o objeto especificado é igual ao objeto atual (herdado de Object.) |
| **Exists** | Determina se List<T> contém os elementos que correspondem às condições definidas pelo predicado especificado. |
| **Find** | As pesquisas para um elemento que corresponde às condições definidas pelo predicado especificado, e retorna a primeira ocorrência de List<T>inteiro. |
| **FindAll** | Recupera todos os elementos que correspondem às condições definidas pelo predicado especificado. |
| **FindLast** | As pesquisas para um elemento que corresponde às condições definidas pelo predicado especificado, e retornam a última ocorrência dentro de List<T>inteiro. |
| **IndexOf(T)** | Pesquisa o objeto especificado e retorna o índice de base zero da primeira ocorrência em List<T>de inteiro. |
| **Insert** | Insere um elemento em List<T> no índice especificado. |
| **InsertRange** | Insere os elementos de uma coleção em List<T> no índice especificado. |
| **Remove** | Remove a primeira ocorrência de um objeto específico de List<T>. |
| **RemoveAll** | Remove todos os elementos que correspondem às condições definidas pelo predicado especificado. |
| **RemoveAt** | Remove o elemento no índice especificado de List<T>. |
| **RemoveRange** | Remove um intervalo dos elementos de List<T>. |
| **Reverse()** | Inverte a ordem dos elementos em List<T>inteiro. |
| **ToArray** | Copia os elementos de List<T> a uma nova matriz. |
| **ToString** | Retorna uma string que representa o objeto atual (herdado de Object.) |

Fonte: Classe... (s.d.).

**Saiba mais**

A partir da versão 2008 do Visual Studio, foi introduzido o LINQ (Consulta Integrada à Linguagem) (MSDN, 2014) (SHARP, 2008), que é um conjunto que amplia os recursos de consulta para a sintaxe da linguagem C# em listas enumeráveis. Para conhecer mais o LINQ, acesse o *site*:

LINQ (Consulta Integrada à Linguagem). [s.d.]. Disponível em: <http://msdn.microsoft.com/pt-br/library/bb397926.aspx>. Acesso em: 27 out. 2014.

Esses são os conceitos fundamentais para a programação em C#. De agora em diante, iremos estudar a POO propriamente dita.

**Resumo**

Nesta unidade foram apresentados os conceitos fundamentais da linguagem C# necessários para o desenvolvimento de aplicações. Foi mostrada a definição da estrutura básica de programas em C# sem ainda utilizar a orientação a objeto. Vimos elementos básicos da programação estruturada, como definição e tipos de variáveis, operadores lógicos, relacionais e matemáticos, desvios condicionais e laços. Essas ferramentas dotam os programas de certa inteligência, possibilitando tomadas de decisões.

Foi dado um destaque para as particularidades da linguagem em si presentes em C#. Conhecemos as características que não estão nas outras linguagens da família de linguagem C, como o C, C++ e o Java. Estudamos também as diferenças da maneira como a entrada de dados é tratada, na qual a variável recebe um valor e este necessita ter o seu tipo convertido; consequentemente, vimos a necessidade de conhecer as classes (funções) de conversão de dados. Em seguida, conhecemos as diferenças na saída no console, como a formatação composta, com sua máscara e os itens de formato que são preenchidos pelos valores dos itens indexados no próprio comando; e vimos também a lista enumerada, em que uma lista pode ser manipulada sem a rigidez de um vetor.

## Exercícios

**Questão 1.** (TRF 2014) Utilize o programa C# abaixo para responder à questão.

Os números à esquerda não fazem parte do programa, apenas indicam os números das linhas.

```
[1] using System;
[2] using System.Collections.Generic;
[3] using System.Linq;
[4] using System.Text;

[5] namespace ConsoleApplication1
[6] {
[7]   public static class ConverteTemp
[8]   {
[9]        public static double CelsiusToFahrenheit(string tempCelsius)
[10]       {
[11]         double celsius = Double.Parse(tempCelsius);
[12]         double fahrenheit = (celsius * 9 / 5) + 32;
[13]         return fahrenheit;
[14]       }

[15]       public static double FahrenheitToCelsius(string tempFahrenheit)
[16]       {
[17]         double fahrenheit = Double.Parse(tempFahrenheit);
[18]         double celsius = (fahrenheit - 32) * 5 / 9;
[19]         return celsius;
[20]       }
[21]
[22]  }

[23]       class TestaConverteTemp
[24]       {
[25]         static void Main()
[26]         {
[27]           Console.WriteLine("Selecione a conversao desejada");
[28]           Console.WriteLine("1. De Celsius para Fahrenheit.");
[29]           Console.WriteLine("2. De Fahrenheit para Celsius.");
[30]           Console.Write(":");

[31]           string opcao = Console.ReadLine();
[32]           double F, C = 0;

[33]           switch (opcao)
[34]           {    case "1":  Console.Write("Digite a temperatura em Celsius: ");
[35]                           F = ConverteTemp.CelsiusToFahrenheit(Console.ReadLine());
[36]                           Console.WriteLine("Temperatura em Fahrenheit: {0:F2}", F);
[37]                           break;

[38]                case "2":  Console.Write("Digite a temperatura em Fahrenheit: ");
[39]                           C = ConverteTemp.FahrenheitToCelsius(Console.ReadLine());
[40]                           Console.WriteLine("Temperatura em Celsius: {0:F2}", C);
[41]                           break;

[42]                default:  Console.WriteLine("Opcao invalida.");
[43]                          break;
[44]           }

[45]           Console.WriteLine("Pressione uma tecla para finalizar.");
[46]           Console.ReadKey();
[47]         }
[48]       }
[49]  }
```

O programa C# apresentado é executado apenas uma vez e finaliza. Para que o programa possa ser executado diversas vezes, até que o usuário digite 0 para finalizá-lo é necessário inserir Console. WriteLine("0.Finaliza."); como mais uma opção do menu e inserir a seguinte instrução de repetição:

A) while (opção!=0) antes do switch, que está na linha 33, com os delimitadores de início e fim { } desta instrução envolvendo as linhas 33 e 44.

B) while (opção!="0") antes do switch que está na linha 33, com os delimitadores de início e fim { } da instrução envolvendo as linhas 33 e 44.

C) for(; ;) { após o delimitador de início de bloco { na linha 26 e uma chave } para fechar o bloco logo após a linha 44. Antes do switch, que está na linha 33, inserir o comando if (opção == "0") break;.

D) do antes do switch, que está na linha 33, com o delimitador de início { da instrução envolvendo as linhas 33 e 44, e finalizando com o delimitador de } while (opção != "0");.

E) for (opção=0; opção < 3; opção++) após o delimitador de início de bloco { na linha 26 com os delimitadores de início e fim {} da instrução envolvendo as linhas 27 e 44.

Resposta correta: alternativa C.

**Análise das alternativas**

A) Alternativa incorreta.

Justificativa: o teste do comando while está errado, pois a variável opção foi declarada como String e não foi convertida para número para ser testada como tal. Para ser testada como String deveria estar como "0" e não como 0, ou, ao ler o valor do teclado, converter para número e somente depois atribuir a variável opção.

B) Alternativa incorreta.

Justificativa: o teste do comando while, depois da leitura da opção escolhida, (linha 32) fará com que não seja possível alterar o valor da variável opção, ficando em loop infinito, pois caso o primeiro seja diferente de 0, jamais haverá outra possibilidade fora do laço de repetição da variável opção receber 0 e interromper a repetição. Para não ocorrer este erro seria necessário que fosse novamente lido um valor para a variável opção antes de encerrar o comando while.

C) Alternativa correta.

Justificativa: mesmo que o comando 'for' não tenha os parâmetros, ele repete até que o teste de para embutido dentro do laço execute o comando break, encerrando assim o laço de repetição.

D) Alternativa incorreta.

Justificativa: esta alternativa está errada por dois motivos: 1) Porque o comando de repetição do está no lugar errado como o while proposto na alternativa B; 2) A cláusula de interrupção do comando do..while teria de ser opção == 0.

E) Alternativa incorreta.

Justificativa: a variável opção não é numérica e contadora do laço de repetição, portanto os parâmetros do comando for estão errados.

**Questão 2**. Veja o programa C# apresentado a seguir:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TESTE
{
    class Program
     {
            static void Main (string[] args)
            {
                    const int a = 7;
                    int b = 1, c = 0, d, e;
                    for (e = 0; e <= a; e++ )
                    {
                           d =b + c;
                           b =c;
                           c = d;
                          Console.WriteLine(c);
                     }
              Console.readKey();
            }
       }
}
```

Qual a saída feita pela execução desse programa?

A) 1 2 3 5 8 13 21 29.

B) 1 2 3 5 7.

C) 1 1 2 3 5 7.

D) 1 1 2 3 5 8 13 21.

E) 1 1 2 3 5 8 13.

**Resolução desta questão na plataforma.**