Unidade 1

1 Variáveis primitivas e Controle de fluxo

Aprenderemos a trabalhar com os seguintes recursos da linguagem C#:

- Declaração, atribuição de valores, casting e comparação de variáveis;
- Controle de fluxo através de if e else;
- Instruções de laço for e while, controle de fluxo com break e continue.

1.1 Declarando e usando variáveis

Dentro de um bloco, podemos declarar variáveis e usá-las.

Em C#, toda variável tem um tipo que não pode ser mudado, uma vez que declarado:

```
tipoDaVariavel nomeDaVariavel;
```

Por exemplo, é possível ter uma idade que vale um número inteiro:

```
int idade;
```

Com isso, você declara a variável idade, que passa a existir a partir deste momento. Ela é do tipo int, que guarda um número inteiro. A partir de agora você pode usá-la, primeiro atribuindo valores.

A linha a seguir é a tradução de “idade deve valer agora quinze”.

```
idade = 15;
```

Comentários em C#

Para fazer um comentário em C#, você pode usar o // para comentar até o final da linha, ou então usar o /* */ para comentar o que estiver entre eles.

```
/* comentário daqui,
ate aqui */


//uma linha de comentário sobre a idade
int idade;
```

Além de atribuir, você pode utilizar esse valor. O código a seguir declara novamente a variável idade com valor 15 e imprime seu valor na saída padrão através da chamada a Console.WriteLine.

```
//declara a idade
int idade;
idade = 15;

// imprime a idade
Console.WriteLine(idade);
```

Por fim, podemos utilizar o valor de uma variável para algum outro propósito, como alterar ou definir uma segunda variável. O código a seguir cria uma variável chamada idadeNoAnoQueVem com valor de idade mais um.

```
//gera a idade no ano seguinte

int idadeNoAnoQueVem;
idadeNoAnoQueVem = idade + 1;
```

No momento que você declara uma variável, também é possível inicializá-la por praticidade:

```
int idade = 15;
```

Você pode usar os operadores +, -, / e * para operar com números, sendo eles responsáveis pela adição, subtração, divisão e multiplicação, respectivamente. Além desses operadores básicos, há o operador % (módulo) que nada mais é que o resto de uma divisão inteira. Veja alguns exemplos:

```
int quatro = 2 + 2;

int tres = 5 - 2;

int oito = 4 * 2;

int dezesseis = 64 / 4;

int um = 5 % 2;
//5 dividido por 2 dá 2 e tem resto 1;
// o operador % pega o resto da divisão inteira
```

Onde testar esses códigos?

Você deve colocar esses trechos de código dentro do método Main.

Isto é, isso deve ficar no miolo do programa. Use bastante Console.WriteLine, dessa forma você pode ver algum resultado, caso contrário, ao executar a aplicação, nada aparecerá. Por exemplo, para imprimir a idade e a idadeNoAnoQueVem podemos escrever o seguinte programa de exemplo:

```
class TestaIdade
{
	static void Main(String[] args)
	{
		// declara a idade
		int idade;
		idade = 15;

		// imprime a idade
		Console.WriteLine(idade);

		// gera uma idade no ano seguinte
		int idadeNoAnoQueVem;
		idadeNoAnoQueVem = idade + 1;

		// imprime a idade
		Console.WriteLine(idadeNoAnoQueVem);
	}
}
```

Representar números inteiros é fácil, mas como guardar valores reais, tais como frações de números inteiros e outros? Outro tipo de variável muito utilizado é o double, que Armazena um número com ponto flutuante.

```
double pi = 3.14;
double x = 5 * 10;
```

O tipo bool armazena um valor verdadeiro ou falso, e só.

```
bool verdade = true;
```

O tipo char guarda um, e apenas um, caractere. Esse caractere deve estar entre aspas simples. Não se esqueça dessas duas características de uma variável do tipo char! Por exemplo, ela não pode guardar um código como ‘’ pois o vazio não é um caractere!

```
char letra = 'a';
Console.WriteLine(letra);
```

1.2 Tipos primitivos e Valores

Esses tipos de variáveis são tipos primitivos do C#: o valor que elas guardam são o real conteúdo da variável. Quando você utilizar o operador de atribuição = o valor será copiado.

```
int i = 5; // i recebe uma cópia do valor 5
int j = i; // j recebe uma cópia do valor de i
i = i + 1; // i vira 6, j continua 5
```

Aqui, i fica com o valor de 6. Mas e j? Na segunda linha, j está valendo 5. Quando i passa a valer 6, será que j também muda de valor? Não, pois o valor de um tipo primitivo sempre é copiado.

Apesar da linha 2 fazer j = i, a partir desse momento essas variáveis não tem relação nenhuma: o que acontece com uma, não reflete em nada com a outra.

Outros tipos primitivos

Vimos aqui os tipos primitivos que mais aparecem. O C# tem outros, que são o byte, short, long e float.

Cada tipo possui características especiais que, para um programador avançado, podem fazer muita diferença.

1.3 Exercícios: Varáveis e tipos primitivos

1) Na empresa onde trabalhamos, há tabelas com o quanto foi gasto em cada mês. Para fechar o balanço do primeiro trimestre, precisamos somar o gasto total. Sabendo que, em Janeiro, foram gastos 15000 reais, em Fevereiro, 23000 reais e em Março, 17000 reais, faça um programa que calcule e imprima o gasto total no trimestre. Siga esses passos:

a) Crie uma classe chamada BalancoTrimestral com um bloco Main, como nos exemplos anteriores;

b) Dentro do Main (o miolo do programa), declare uma variável inteira chamada gastosJaneiro e inicialize-a com 15000;

c) Crie também as variáveis gastosFevereiro e gastosMarco, inicializando-as com 23000 e 17000, respectivamente, utilize uma linha para cada declaração;

d) Crie uma variável chamada gastosTrimestre e inicialize-a com a soma das outras 3 variáveis:

```
int gastosTrimestre = gastosJaneiro + gastosFevereiro + gastosMarco;
```

e) Imprima a variável gastosTrimestre.

2) (opcional) Adicione código (sem alterar as linhas que já existem) no programa a seguir para imprimir o resultado:

Resultado: 15, 15.1, y, false

```
class ExercicioSimples
{
	static void Main(String[] args)
	{
		int i = 10;
		double d = 5;
		char c = 't';
		bool b = true;
		// imprime concatenando diversas variáveis

		Console.WriteLine("Resultado: " + i + ", " + d + ", " + c
+ ", " + b);

	}
}
```

1.4 - Discussão em aula: convenções de código e código legível

Discuta com seus colegas sobre convenções de código C#. Por que existem? Por que são importantes? Discuta também as vantagens de se escrever código fácil de ler e se evitar comentários em excesso.

1.5 - Casting

Alguns valores são incompatíveis se você tentar fazer uma atribuição direta. Enquanto um número real costuma ser representado em uma variável do tipo double, tentar atribuir ele a uma variável int não funciona porque é um código que diz: **“i deve valer d”**, mas não se sabe se d realmente é um número inteiro ou não.

```
double d = 3.1415;
int i = d; // não compila
```

O mesmo ocorre no seguinte trecho:

```
int i = 3.14;
```

O mais interessante, é que nem mesmo o seguinte código compila:

```
double d = 5; // ok, o double pode conter um número inteiro
int i = d; // não compila
```

Apesar de 5 ser um bom valor para um int, o compilador não tem como saber que valor estará dentro desse double no momento da execução. Esse valor pode ter sido

digitado pelo usuário, e ninguém vai garantir que essa conversão ocorra sem perda de valores.

Já no caso a seguir, é o contrário:

```
int i = 5;
double d2 = i;
```

O código acima compila sem problemas, já que um double pode guardar um número com ou sem ponto flutuante. Todos os inteiros representados por uma variável do tipo int podem ser guardados em uma variável double, então não existem problemas no código acima.

Às vezes, precisamos que um número quebrado seja arredondado e armazenado num número inteiro. Para fazer isso sem que haja o erro de compilação, é preciso ordenar que o número quebrado seja **moldado (casted)** como um número inteiro. Esse processo recebe o nome de **casting**.

```
double d3 = 3.14;
int i = (int)d3;
```

O casting foi feito para moldar a variável d3 como um int. O valor de i agora é 3.

O mesmo ocorre entre valores int e long.

```
long x = 10000;
int i = x; // nao compila, pois pode estar perdendo informação
```

E, se quisermos realmente fazer isso, fazemos o casting:

```
long x = 10000;
int i = (int)x;
```

Casos não tão comuns de casting e atribuição

Alguns **castings** aparecem também:

```
float x = 0.0;
```

O código acima não compila pois todos os literais com ponto flutuante são considerados Double pelo C#. E float não pode receber um double sem perda de informação, para fazer isso funcionar podemos escrever o seguinte:

```
float x = 0.0f;
```

A letra f, que pode ser máiscula ou minúscula, indica que aquele literal deve ser tratado como float.

Outro caso, que é mais comum:

```
double d = 5;
float f = 3;
float x = f + (float)d;
```

Você precisa do casting porque o C# faz as contas e vai armazenando sempre no maior tipo que apareceu durante as operações, no caso o double.

E uma observação: no mínimo, o C# armazena o resultado em um int, na hora de fazer as contas.

Até casting com variáveis do tipo char podem ocorrer. O único tipo primitivo que não pode ser atribuído a nenhum outro tipo é o bool.

Castings possíveis

Abaixo estão relacionados todos os casts possíveis na linguagem C#, mostrando a conversão **de** um valor **para** outro. A indicação **Impl.** quer dizer que aquele cast é implícito e automático, ou seja, você não precisa indicar o cast explicitamente (lembrando que o tipo bool não pode ser convertido para nenhum outro tipo).

PARA: / DE:	byte	short	char	int	long	float	double
byte	----	*Impl.*	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
short	(byte)	----	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
char	(byte)	(short)	----	*Impl.*	*Impl.*	*Impl.*	*Impl.*
int	(byte)	(short)	(char)	----	*Impl.*	*Impl.*	*Impl.*
long	(byte)	(short)	(char)	(int)	----	*Impl.*	*Impl.*
float	(byte)	(short)	(char)	(int)	(long)	----	*Impl.*
double	(byte)	(short)	(char)	(int)	(long)	(float)	----

1.6 - O If-Else

A sintaxe do if no C# é a seguinte

```
if (condicaoBooleana)
{
	codigo;
}
```

Uma **condição booleana** é qualquer expressão que retorne true ou false. Para isso, você pode usar os operadores <, >, <=, >= e outros. Um exemplo:

```
int idade = 15;
if (idade < 18)
{
	Console.WriteLine("Não pode entrar");
}
```

Além disso, você pode usar a cláusula else para indicar o comportamento que deve ser executado no caso da expressão booleana ser falsa:

```
int idade = 15;
if (idade < 18)
{
	Console.WriteLine("Não pode entrar");
}
else
{
	Console.WriteLine("Pode entrar");
}
```

Você pode concatenar expressões booleanas através dos operadores lógicos **"E"** e **"OU"**. O **"E"** é representado pelo & e o **"OU"** é representado pelo |.

```
int idade = 15;
bool amigoDoDono = true;
if (idade < 18 & amigoDoDono == false)
{
	Console.WriteLine("Não pode entrar");
}
else
{
	Console.WriteLine("Pode entrar");
}
```

Esse código poderia ficar ainda mais legível, utilizando-se o operador de negação, o !. Esse operador transforma uma expressão booleana de false para true e vice versa.

```
int idade = 15;
bool amigoDoDono = true;
if (idade < 18 & !amigoDoDono)
{
      Console.WriteLine("Não pode entrar");
}
else
{
      Console.WriteLine("Pode entrar");
}
```

Repare na linha 3 que o trecho amigoDoDono == false virou !amigoDoDono. **Eles têm o mesmo valor.**

Para comparar se uma variável tem o mesmo valor que outra variável ou valor, utilizamos o operador ==.

Repare que utilizar o operador = vai retornar um erro de compilação, já que o operador = é o de atribuição.

```
int mes = 1;
if (mes == 1)
{
      Console.WriteLine("Você deveria estar de férias");
}
```

&& ou &

Em alguns livros, logo será apresentado a você dois tipos de operadores de OU e de E. Você realmente não precisa saber distinguir a diferença entre eles por enquanto.

O que acontece é que os operadores && e || funcionam como seus operadores irmãos, porém eles funcionam da maneira mais rápida possível: quando percebem que a resposta não mudará mais, eles param de verificar as outras condições booleanas. Por isso, eles são chamados de operadores de curto circuito (short circuit operators).

```
if (true | algumaCoisa)
{
      // ...
}
```

O valor de algumaCoisa será analisado, nesse caso. Repare que não precisaria, pois já temos um true. true ou qualquer outra coisa, dá sempre true.

```
if (true || algumaCoisa)
{
      // ...
}
```

Neste caso o algumaCoisa não será analisado. Pode não fazer sentido ter as duas opções, mas em alguns casos é

interessante e útil usar um ou outro, além de dar diferença no resultado. Veremos mais adiante em outros capítulos.

1.7 - O While

O while é um comando usado para fazer um **laço (loop)**, isto é, repetir um trecho de código algumas vezes.

A idéia é que esse trecho de código seja repetido enquanto uma determinada condição permanecer verdadeira.

```
int idade = 15;
while (idade < 18)
{
	Console.WriteLine(idade);
	idade = idade + 1;
}
```

O trecho dentro do bloco do while será executado até o momento em que a condição idade < 18 passe a ser falsa. E isso ocorrerá exatamente no momento em que idade == 18, o que não o fará imprimir 18.

```
int i = 15;
while (i < 10)
{
	Console.WriteLine(i);
	i = i + 1;
}
```

Já o while acima imprime de 0 a 9.

1.8 - O For

Outro comando de **loop** extremamente utilizado é o for. A idéia é a mesma do while: fazer um trecho de código ser repetido enquanto uma condição continuar verdadeira. Mas, além disso, o for isola também um espaço para inicialização de variáveis e o modificador dessas variáveis. Isso faz com que fiquem mais legíveis, as variáveis que são relacionadas ao loop:

```
for (inicializacao; condicao; incremento)
{
	codigo;
}
```

Um exemplo é o a seguir:

```
for (int i = 0; i < 10; i = i + 1)
{
	Console.WriteLine("olá!");
}
```

Repare que esse for poderia ser trocado por:

```
int i = 0;
while (i < 10)
{
	Console.WriteLine("olá!");
	i = i + 1;
}
```

Porém, o código do for indica claramente que a variável i serve, em especial, para controlar a quantidade de laços executados. Quando usar o for? Quando usar o while? Depende do gosto e da ocasião.

Pós incremento ++

i = i + 1 pode realmente ser substituído por i++ quando isolado, porém, em alguns casos, temos essa instrução envolvida em, por exemplo, uma atribuição:

```
int i = 5;
int x = i++;
```

Qual é o valor de x? O de i, após essa linha, é 6.

O operador ++, quando vem após a variável, retorna o valor antigo, e incrementa (pós incremento), fazendo x valer 5.

Se você tivesse usado o ++ antes da variável (pré incremento), o resultado seria 6:

```
int i = 5;
int x = ++i; // aqui x valera 6
```

1.9 - Controlando loops

Apesar de termos condições booleanas nos nossos laços, em algum momento podemos decidir parar o loop por algum motivo especial, sem que o resto do laço seja executado.

```
for (int i = 0; i < 100; i++)
{
	if (i % 19 == 0)
	{
		Console.WriteLine("Achei um número divisível por 19 entre x
	e y");
		break;
	}
}
```

O código acima vai percorrer os números e parar quando encontrar um número divisível por 19, uma vez que foi utilizada a palavra chave break.

Da mesma maneira, é possível obrigar o loop a executar o próximo laço. Para isso usamos a palavra chave continue.

```
for (int i = 0; i < 100; i++)
{
      if (i > 50 && i < 60)
      {
            continue;
      }
      Console.WriteLine(i);
}
```

O código acima não vai imprimir alguns números. (Quais exatamente?)

1.10 - Escopo das variáveis

No C#, podemos declarar variáveis a qualquer momento. Porém, dependendo de onde você as declarou, ela vai valer de um determinado ponto a outro.

```
//aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
```

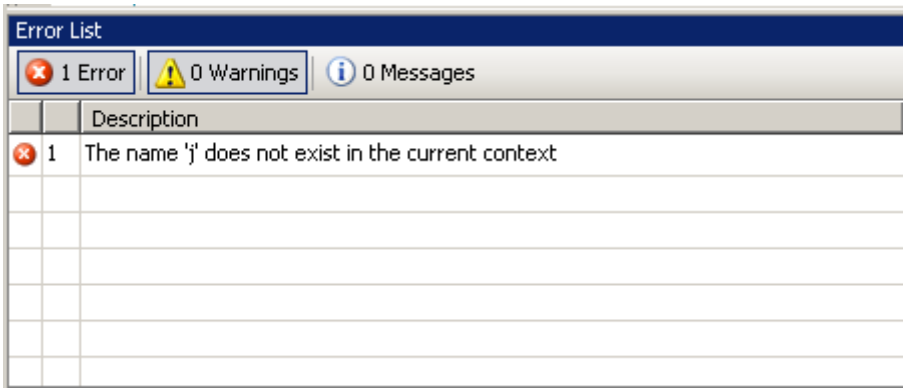
O **escopo da variável** é o nome dado ao trecho de código em que aquela variável existe e que é possível acessá-la.

Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro **só valem até o fim daquele bloco**.

```
//aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
while (condicao)
{
      // o i ainda vale aqui
      int j = 7;
      // o j passa a existir
}
// aqui o j não existe mais, mas o i continua a valer

j = 8;
```

No bloco acima, a variável j pára de existir quando termina o bloco onde ela foi declarada. Se você tentar acessar uma variável fora de seu escopo, ocorrerá um erro de compilação.



O mesmo vale para um if:

```
if (algumBooleano)
{
      int i = 5;
}
else
{
      int i = 10;
}

Console.WriteLine(i); // cuidado!
```

Aqui a variável i não existe fora do if e do else! Se você declarar a variável antes do if, vai haver outro erro de compilação: dentro do if e do else a variável está sendo redeclarada! Então o código para compilar e fazer sentido fica:

```
int i;
if (algumBooleano)
{
      i = 5;
}
else
{
      i = 10;
}
Console.WriteLine(i);
```

Uma situação parecida pode ocorrer com o for:

```
for (int i = 0; i < 10; i++)
{
      Console.WriteLine("olá!");
}
Console.WriteLine(i); // cuidado!
```

Neste for a variável i morre ao seu término, não podendo ser acessada de fora do for, gerando um erro de compilação. Se você realmente quer acessar o contador depois do loop terminar, precisa de algo como:

```
int i;
for (i = 0; i < 10; i++)
{
	Console.WriteLine("olá!");
}
Console.WriteLine(i);
```

1.11 - Um bloco dentro do outro

Um bloco também pode ser declarado dentro de outro. Isto é, um if dentro de um for, ou um for dentro de um for, algo como:

```
while (condicao)
{
	for (int i = 0; i < 10; i++)
	{
		// código
	}
}
```

1.12 - Um pouco mais...

1) Vimos apenas os comandos mais usados para controle de fluxo. O C# ainda possui o do..while e o switch. Pesquise sobre eles e diga quando é interessante usar cada um deles.

2) O que acontece se você tentar dividir um número inteiro por 0? E por 0.0?

3) Existem outros operadores, como o %, <<, >>. Descubra para que servem.

4) Além dos operadores de incremento, existem os de decremento, como --i e i--. Além desses, você pode usar instruções do tipo i += x e i -= x, o que essas instruções fazem? Teste.

1.13 - Exercícios: Fixação de sintaxe

Mais exercícios de fixação de sintaxe. Para quem já conhece um pouco de C# pode ser muito simples, mas recomendamos fortemente que você faça os exercícios para se acostumar com erros de compilação, mensagens do compilador, convenção de código, etc...

Apesar de extremamente simples, precisamos praticar a sintaxe que estamos aprendendo. Para cada exercício, crie um novo arquivo, e declare aquele estranho cabeçalho, dando nome a uma classe e com um método Main dentro dele:

```
class ExercicioX
{
	static void Main(String[] args)
	{
		// seu exercicio vai aqui
	}
}
```

Não copie e cole de um exercício já existente! Aproveite para praticar.

1) Imprima todos os números de 150 a 300.

2) Imprima a soma de 1 até 1000.

3) Imprima todos os múltiplos de 3, entre 1 e 100.

4) (opcional) Imprima os primeiros números da série de Fibonacci até passar de 100. A série de Fibonacci é a seguinte: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc... Para calculá-la, o primeiro e segundo elementos valem 1, daí por diante, o n-ésimo elemento vale o (n-1)-ésimo elemento somando ao (n-2)-ésimo elemento (ex: 8 = 5 + 3).

5) (opcional) Escreva um programa que, dada uma variável x (com valor 180, por exemplo), temos y de acordo com a seguinte regra:

- se x é par, y = x / 2
- se x é impar, y = 3 * x + 1
- imprime y
- O programa deve então jogar o valor de y em x e continuar até que y tenha o valor final de 1. Por exemplo, para x = 13, a saída será:

40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

Imprimindo sem pular linha

Um detalhe importante do método que estamos usando até agora é que uma quebra de linha é impressa toda vez que chamado. Para não pular uma linha usamos o método a seguir:

```
Console.Write("Flamengo campeão de 2010");
```