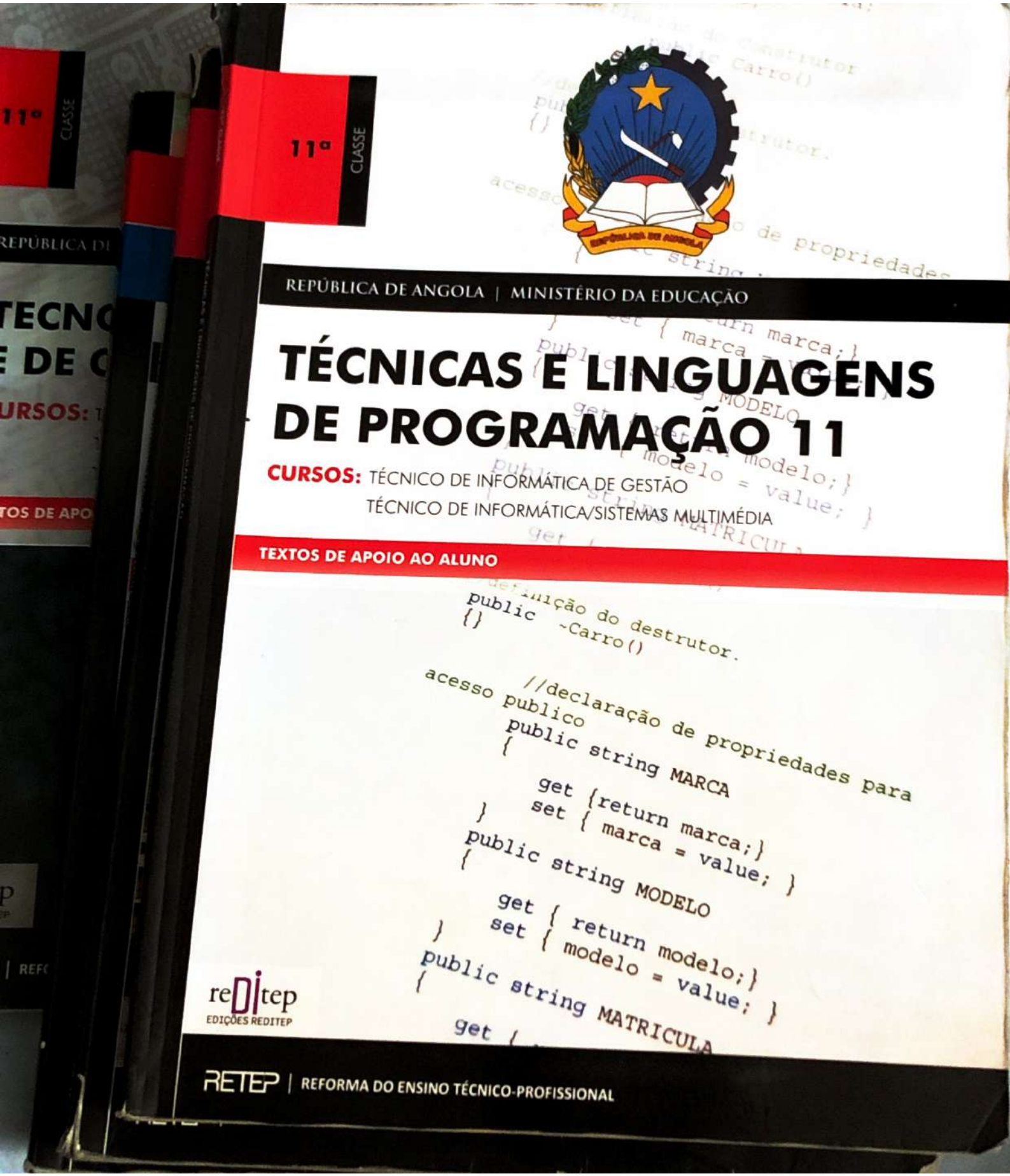

11ª CLASSE
REPÚBLICA DE ANGOLA | MINISTÉRIO DA EDUCAÇÃO
TÉCNICAS E LINGUAGENS DE PROGRAMAÇÃO 11
CURSOS: TÉCNICO DE INFORMÁTICA DE GESTÃO
TÉCNICO DE INFORMÁTICA/SISTEMAS MULTIMÉDIA
TEXTOS DE APOIO AO ALUNO
reDitep
EDIÇÕES REDITEP
RETEP | REFORMA DO ENSINO TÉCNICO-PROFISSIONAL

Capítulo 1

PROGRAMAÇÃO NUMA LINGUAGEM ORIENTADA AOS OBJECTOS

CONTEÚDO

1.1 Características da programação orientada aos objectos.

1.2 Introdução à programação numa linguagem orientada por objectos.

1.3 Introdução à programação numa linguagem visual.

1.4 Componentes básicos da programação numa linguagem visual.

OBJECTIVOS

- Conhecer as características de interpretação e desenvolvimento de software numa linguagem orientada por objectos.
- Identificar vantagens na utilização das ferramentas de programação.
- Definir os conceitos básicos de classe e objectos.
- Definir propriedades, funções membro e comportamentos.
- Conhecer as características de desenvolvimento de software numa linguagem orientada por objectos.
- Conceitos básicos de programação na linguagem C#.
- Conhecer as características e desenvolvimento de software numa linguagem orientada por objectos em ambiente visual.
- Utilização das ferramentas de programação visual.

1.1 CARACTERÍSTICAS DA PROGRAMAÇÃO ORIENTADA AOS OBJECTOS

1.1.1 INTRODUÇÃO

A programação orientada por objectos (POO) é um paradigma de programação que se centra na noção de objecto, ao contrário da programação tradicional que é centrada no conceito de procedimento. A POO utiliza objectos e não algoritmos como bloco lógico fundamental. Um objecto, neste contexto, representa entidades (por exemplo, pessoas, coisas ou processos) do mundo real. A POO é essencialmente um estilo de programação, de certa forma independente da linguagem de programação. Pode-se programar em C usando objectos e outros conceitos OO, assim como se pode usar uma linguagem OO, como o C++, sem recorrer a qualquer conceito de POO. No entanto, muitos dos conceitos de POO só são suportados por linguagens de programação orientadas por objectos.

1.1.2 CONCEITO DE PROGRAMAÇÃO ORIENTADA POR OBJECTOS

A programação orientada por objectos (POO) é um método ou forma de desenvolver código e consequentemente *software*. Enquanto nas linguagens procedimentais, como o caso da Linguagem C, desenvolvemos código e funções identificando as variáveis recorrendo somente a um comportamento, numa linguagem orientada por objectos podemos ir mais longe e representar qualquer objecto do dia-a-dia.

Com a programação orientada por objectos é possível desenvolver aplicações mais robustas e permitem ao programador uma maior reutilização de código e consequente baixa probabilidade de falha na produção do *software*.

Analisemos as principais diferenças entre as linguagens procedimentais e as linguagens orientadas por objectos. Por exemplo, em linguagem C, para representar um objecto "Mesa" poderia criar uma estrutura e as variáveis "Tampo", "Altura", "Largura", "Comprimento". Não deixa de ser um conjunto de quatro variáveis que identifica a "Mesa". Por cada tipo de mesa, em que o número de variáveis fosse diferente, teríamos de definir uma nova estrutura e criar tudo de novo.

Numa abordagem de POO poderemos criar uma classe que identifique a "Mesa" e reutilizá-la as vezes que entender no programa e se a "Mesa" for diferente, basta herdar as propriedades da anterior e cria uma nova classe.

Parece confuso na teoria, mas verá que na prática é mais simples!

Vejamos outro exemplo:

Um motociclo é um objecto da vida real e este tem um conjunto de características e comportamentos ou acções. Às características vamos chamar de propriedades e às acções de métodos. Assim, temos como propriedades a cor, a cilindrada, a marca, o modelo, o ano de fabrico, entre outras. Como métodos temos acelerar, travar, ligar faróis, buzina, entre outros.

No entanto, para este objecto em causa, o motociclo só acelera se nós rodarmos o punho do acelerador e neste caso temos um evento que para funcionar, poderá executar um método da classe.

No entanto, analisemos que o facto de eu rodar o punho do acelerador, se o motociclo não estiver a trabalhar, a acção de rodar o punho do acelerador, não irá gerar nenhuma reacção do motociclo a não ser que tenha já rodado a chave (outro evento) e o motociclo estivesse a trabalhar (estado do objecto: ligado ou desligado).

Como é de esperar, o motociclista não está preocupado com o ciclo de eventos e resultados que implica rodar a chave de ignição (estado da bateria, bomba de combustível, rodar chave e dar circuito à ignição, gerar a ignição e queimar o combustível, entre outras centenas de eventos que decorrem) mas sim com o facto de o motociclo estar a trabalhar.

Podemos então perceber que, para aceder às propriedades e alterar o estado do objecto, teremos de aceder através dos métodos e eventos. A esta característica chamamos de encapsulamento. Este comportamento garante-nos que as propriedades só são alteradas pelos seus métodos e não do "exterior" (na linha de código que estamos a invocar). A esta forma de mascara entre a classe e como a usamos é definida como encapsulamento.

Esta visão simples das coisas é entendida como abstracção, que é uma das principais funcionalidades que obtemos em usar esta forma de programação. No exemplo do motociclo, quando eu tenho necessidade de me deslocar, não temos o hábito de explicar ao pormenor como "retirar a chave e colocar na ignição, verificar carga da bateria e combustível, rodar a chave, accionar a embraiagem e usar a caixa de velocidades, engrenar a primeira velocidade, controlar a velocidade, controlar a direcção com o guiador, ...", mas dizemos sim "vou de motociclo até ao trabalho". No fundo, o programador detalha a sua classe com tudo o que necessita para representar o objecto e não tem de carregar o código com instruções adicionais e ter de as repetir sempre que a função é repetida.

1.2 INTRODUÇÃO À PROGRAMAÇÃO NUMA LINGUAGEM ORIENTADA POR OBJECTOS

1.2.1 DEFINIÇÃO DE CLASSE, OBJECTOS E FUNCIONALIDADES DE POO

Classe

Uma classe é uma estrutura de programação que descreve determinado objecto, definindo propriedades e métodos. Uma classe é somente a definição de onde serão armazenados os dados e como serão invocados (que métodos vamos invocar). Podemos considerar que a classe é o "molde" de dados, depois será instanciada por cada objecto que necessite de armazenar.

Exemplo

No exemplo que abordamos sobre o motociclo, seria na classe que definíamos as suas propriedades (cor, cilindrada, marca, modelo, ano de fabrico, entre outras), bem como métodos (acelerar, travar, ligar faróis, buzina, entre outros) e eventos como o *construtor* e *destructor* que fazem com que o nosso objecto exista ou seja eliminado do nosso programa.

Objecto

O objecto é uma "variável" do tipo da classe. Parece um pouco estranha esta definição mas, na realidade, um objecto é uma instância da classe. Assim, por cada objecto diferente, teremos de instanciar a classe e utilizar os métodos para aceder e alterar o seu estado (aceder à propriedade). Para instanciarmos um objecto, recorremos ao construtor da classe (método específico para gerar um objecto de qualquer classe) e recorremos ao *destructor* para eliminar a instância do objecto.

Propriedades ou variáveis membro

As propriedades são as variáveis que permitirão guardar os dados do objecto. Estes podem ser inteiros, reais, *strings*, estruturas e inclusive outros objectos ou apontadores de objectos. Em geral, são privadas e acedidas pelos métodos.

Métodos ou funções membro

Os métodos são procedimentos declarados na classe que permitirá aos programadores alterar o estado do objecto. São funções que executam todo o código necessário para a gestão do objecto e integração com outros componentes do programa.

Encapsulamento

O encapsulamento distingue-se pela particularidade de colocar uma espécie de cápsula nas propriedades dos objectos e somente deixar aceder aos mesmos através dos métodos.

Por exemplo, se desejar mudar o estado do motociclo de desligado para ligado, teremos de recorrer ao evento de ignição (que irá mudar o estado internamente do objecto para motociclo ligado).

Assim, podemos definir que o encapsulamento será uma protecção que o objecto garante às propriedades e aos métodos, impedindo que se aceda de forma directa e somente deixará aceder às propriedades através de métodos públicos. A forma comum, na maioria das linguagens de programação OO é utilizada a expressão *private*.

Herança

A herança pode ser considerada com uma forma de reutilização de código. De uma forma rápida, definimos como a possibilidade de particularizar algumas classes tendo a definição de outras como base.

Por exemplo, podemos ter uma classe que define "pessoa" e depois ter a classe "aluno", que é criada e tem as propriedades da classe "mãe"; poderá ser particularizada, quer nas propriedades quer nos métodos, por características que dizem respeito somente à classe "filha".

Polimorfismo

Normalmente, o polimorfismo está interligado com a herança da classe. Quando existe a necessidade de criar uma classe herdada, podemos redefinir o método com o mesmo nome da classe que o herda mas com resultados diferentes. O método tem os mesmos nomes mas com comportamento diferentes tem origem na própria definição: "poli" – múltiplas e "morfismo" – formas.

1.2.2 EXEMPLO PRÁTICO DE APLICAÇÃO DE UMA CLASSE EM C#

O desafio que se impõe será demonstrar na prática como declaramos uma classe e como aceder aos dados membro e métodos.

Analisemos o seguinte exemplo de uma classe, utilizando linguagem C#.

```
class Carro
    {
        //declaração das propriedades
        string marca;
        string modelo;
        string matricula;
        int ano;

        //definição do Construtor
        public Carro()
        {}

//definição do destrutor.
public ~Carro()
{}
        //declaração de propriedades para
acesso publico
        public string MARCA
        {
            get {return marca;}
            set { marca = value; }
        }
        public string MODELO
        {
            get { return modelo;}
            set { modelo = value; }
        }
        public string MATRICULA
        {
            get { return matricula; }
            set { matricula = value; }
        }
        public int ANO
        {
            get { return ano; }
            set { ano = value; }
        }

//funções membro ou método de uma classe
        public void AlteraAno(int novoAno)
        {
            ano = novoAno;
        }
    }
```

```
        // esta função tipo é da classe
laranja e para executar o da base temos de
colocar "base.tipo()" de forma a executar o
tipo da base e o que estiver aqui nesta função
        public override string Tipo()
        {
            return base.Tipo() + nome;
        }
        public override string Descascar()
        {
            return texto + nome;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Fruta f1 = new Banana(65,15);
            Fruta f2 = new Laranja(78,20);
            Fruta f3 = new Fruta(100,100);
            Console.WriteLine(f1.Tipo());
            Console.WriteLine(f2.Tipo());
            Console.WriteLine(f3.Tipo());
            Console.WriteLine(((Banana)f1).NOME);
            Console.WriteLine(f1.Descascar());
            Console.WriteLine(f2.Descascar());
            Console.WriteLine(f3.Descascar());
            Console.ReadKey();
        }
    }
```

O resultado final:

```
Fruta, com peso de 65 e calorias 15 é: Banana
Fruta, com peso de 78 e calorias 20 é: Laranja
Fruta, com peso de 100 e calorias 100 é: Fruta
Estou a descascar uma: Banana
Estou a descascar uma: Laranja
Estou a descascar uma: Fruta
```

Como podem observar no exemplo anterior, criamos inicialmente uma classe `Fruta`, que tem como dados membro o peso e as calorias. Depois definimos dois construtores, `public Fruta()` e também `public Fruta(int peso, int calorias)`, o primeiro sem atributos (inicializa os dados membro com valores nulos) e o segundo com dois argumentos inteiros, um para peso e outro para calorias. Chamamos a esta funcionalidade de *overloading*, neste caso aplicado ao construtor. Iremos observar que esses dados membros serão utilizados pelas classes derivadas.

Por outro lado estamos perante um exemplo de polimorfismo. A função `Tipo()` foi declarada na classe mãe e também nas classes derivadas. Estamos perante o mesmo nome da função mas tem comportamentos diferentes em cada uma das declarações. Além de que, neste caso específico, invocamos a função da classe base na classe derivada, bastando colocar `base.Tipo()` para chamarmos a função.

No entanto, sempre que pretendemos usar a função da classe base modificada nas classes derivadas, teremos de, na classe base, usar a expressão virtual e nas classes derivadas a expressão *override* para que o compilador saiba onde executar a função membro.

Iremos de seguida analisar com mais pormenor a forma de desenvolvimento da linguagem C#, aplicando os conceitos da programação orientada por objectos.

1.2.7 CLASSES E MÉTODOS ABSTRACTOS

Uma classe abstracta não pode ser instanciada (só classes derivadas podem ser instanciadas). Por outro lado, o método abstracto não tem implementação na classe em que é declarado (só em classes derivadas).

- A classe tem de ser abstracta.
- É implicitamente virtual (mas não leva virtual).

Analisemos o seguinte exemplo:

```
public abstract class Shape
    {
    public abstract void Resize(double factor);
    public void DoubleSize() { Resize(2.0); }
    }
public class Box: Shape
    {
    public override void Resize(double factor)
        { ... }
}
```

1.2.6 DERIVAÇÃO DE CLASSES HERANÇA E POLIMORFISMO

Um dos principais objectivos da programação orientada por objectos é a criação de classes, objectos e sua derivação.

Quando usamos uma classe base, ou também chamada de classe mãe, e derivamos numa outra classe que chamamos de classe derivada, obtemos uma melhor definição do código gerado. A derivação de classe pode não ficar pelo primeiro nível mas sim acontecer e derivando para outros níveis, criando assim uma hierarquia de classes. Por exemplo, podemos definir uma classe veículo como classe base e depois derivar nas classes terrestre, aéreo e anfíbio, por sua vez a classe terrestre pode derivar por tipo de transporte como comboio, automóvel, motociclo, entre outros.

A derivação de classes com herança das classes base e com as suas características próprias, juntamente com a possibilidade de criarmos e utilizarmos métodos em todas essas classes constitui a essência da programação orientada por objectos.

Durante a derivação de classes vamos ter necessidade de recorrer a herança e polimorfismo.

Observemos o primeiro exemplo de herança com recursos a classes que demonstre de forma clara o conceito.

Temos uma classe `rectangulo` que representa um rectângulo com dois dados membro sendo eles o `comprimento` e a `largura`. Com esta classe a servir de base, iremos criar a classe derivada `tanque`. A classe rectângulo tem ainda duas funções membro, a `setdados()` para definir o valor do comprimento e da largura e a `getarea()` que devolve um *float* com o valor da área.

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

namespace Manual
{
    class Program
    {
        public class rectangulo {
            float comprimento;
            float largura;
            public rectangulo()
            {}
            public virtual void setdados(float cp, float lg)
            {
                comprimento = cp;
                largura = lg;
            }
            public virtual float getarea()
            {
                return (comprimento * largura);
            }
        }
        public class tanque:rectangulo
        {
            float altura;
            public void setaltura(float a) {altura=a;}
            public float getaltura(){return altura;}
            public override float getarea()
            {
                return base.getarea();
            }
            public override void setdados(float c, float l){
                base.setdados(c,l);
            }

        }
        static void Main(string[] args)
        {
        tanque t1=new tanque();
        t1.setaltura(3);
        t1.setdados(4,5);
        Console.WriteLine(" o volume do tanque=");
        Console.WriteLine(t1.getaltura()*t1.getarea());
        Console.ReadKey();
        }
    }
}
```

O resultado final:

```
O volume do tanque = 60
```

Como podem observar, ao criarmos o objecto tanque, é executado o "construtor" do tanque e também do rectângulo, e também acedemos às funções membro de ambos.

Desta forma conseguimos criar mais objectos derivados da classe rectângulo e evitar a duplicação de código.

Ao executarmos `tanque t1=new tanque();` estamos a criar um objecto `t1` do tipo tanque. Como invocamos o "construtor" de `tanque`, o sistema irá invocar também o "construtor" de `rectangulo`. De seguida, podemos aceder

1.2.3 DECLARAR A CLASSE

Em linguagem C#, os comentários são iniciados com "//" e os blocos de código com "{" que significa abertura de código e "}" que significa conclusão do bloco de código (no capítulo específico iremos analisar a sintaxe de programação da Linguagem C#).

No exemplo anterior, iniciamos a definição da classe com o seu nome "Carro" e de seguida a declaração dos dados membro (em C#, se não for indicado o contrário, é interpretado com propriedades privadas e não tem a necessidade de usar a expressão "*private*".

De seguida são declarados o *construtor* e *destructor* da classe (em C# não é necessário declarar o *destructor*, a não ser que queiramos desenvolvê-lo). A declaração do *construtor* será invocada sempre que criar um novo objecto carro. Assim, podemos utilizar o seguinte código:

```
Carro meuCarro = new Carro();
```

1.2.4 O CONSTRUTOR E O DESTRUCTOR

Desta forma, declaramos um objecto `meuCarro` e é invocado o construtor para gerar o objecto em memória, garantindo todos os componentes da definição da classe (propriedades, métodos e eventos). A chamada do construtor é feita com a expressão `new Carro();` Como vimos, a criação de um objecto de uma classe faz-se simplesmente declarando uma variável dessa classe. Com esta declaração, invocando o construtor, criamos um objecto do tipo `Carro`, com todos os seus elementos. No entanto, podemos ter construtores com argumentos, o que possibilita criarmos um objecto com valores iniciais por exemplo.

Vejamos um exemplo:

```
//definição do Construtor
        public Carro(string mar, string mod,
string mat, int a)
        {
      marca=mar;
      modelo=mod;
      matricula=mat;
      ano=a;
}
```


Os "Destrutores" são métodos com nome da classe precedido de "~", usados para "destruir" um objecto que vai ser libertado de memória.

Características:

- São chamados pelo *garbage collector*.
- Um objecto pode ser libertado de memória a partir do momento em que não pode ser usado por nenhuma parte do código.
- Não têm parâmetros nem valor de retorno (mas não levam `void`).
- Correspondem ao método `Finalize` no CLR, implementado desde `System.Object`.

Assim, invocando os construtores estamos a criar um objecto com valores iniciais que são passados pelo construtor. Desta forma, no nosso programa principal, teremos de instanciar a classe da seguinte forma:

```
Carro meuCarro = new Carro("Toyota","Land Cruiser","XXX-
02-34",2000);
```

Com as declarações das propriedades, podemos aceder aos conteúdos dos dados membro desde o exterior da definição da classe. Neste exemplo têm código desenvolvido para todas as propriedades. Assim, desde o programa principal, podemos invocar `meuCarro.MARCA = "Toyota";`

Por último, temos uma declaração de um método que servirá para mudar o "Ano de fabrico" do veículo `public void AlteraAno(int novoAno)` que quando invocado irá mudar o conteúdo da propriedade, desta forma: `meuCarro.AlteraAno(2003);`

1.2.5 VARIÁVEIS MEMBRO DO TIPO *STATIC*

As variáveis em C# poderão ser globais e locais. Nas variáveis membro de uma classe também podemos criar um tipo *Static* que se comporta de maneira diferente. Enquanto as outras variáveis membro são específicas para cada objecto, quando declaramos uma variável `static`, esta é partilhada por todos os objectos.

Por exemplo, se pretender contar o número de objectos criados, podemos declarar uma variável "`static int n_objectos;`" e sempre que invoco o construtor podemos incrementar este inteiro em uma unidade (de reforçar que na Framework .NET, todos os componentes são objectos, *strings*, inteiros, caixas de texto, são todos objectos).

funcionam as variáveis. Iremos proceder a um pequeno teste. No "exercicio02" antes da última linha `Console.ReadKey()`, vamos acrescentar o código:

```
//Declaração de variáveis
string texto;
int x;
int y = 3;
double d = 2.5;
//Escrever texto para a consola
x = 10;
Console.WriteLine("O Valor de X+Y é: " + (x+y));
Console.WriteLine("O Valor de Y é: " + y);
Console.WriteLine("O Valor de d é: {0}", d);
Console.WriteLine("O Valor da soma de {0} com {1} é = a
{2}",x,y,(x+y));

Console.WriteLine("O Valor da multiplicação de {0} com
{1} é = a {2}", x, y, x*y);
Console.WriteLine("O Valor da divisão de {0} com {1} é =
a {2}", x, y, (float)x /y);
Console.WriteLine("O resto da divisão inteira de {0} com
{1} é = a {2}", x, y, x % y);
//Leitura de Dados da consola. Só lê strings. Se
necessitarmos, teremos de converter
texto = Console.ReadLine();
//writeline com caracteres de escape, neste caso a "\".
Console.WriteLine("O texto digitado foi: " + texto);
//Ler inteiros
Console.WriteLine("Digite o valor de X");
try
  {
  x = Convert.ToInt32(Console.ReadLine());
  //também dá para utilizar o x=int.Parse
(Console.ReadLine ());
  }
catch { }
//para tratar excepções, posso usar o try e o catch para
evitar os erros (assim não pára de introduzir texto em
vez de inteiro.
Console.WriteLine("O Valor de X é: {0}", x);
//Usar o ReadKey
Console.ReadKey();
```

Mais uma vez, vamos analisar o código digitado. Na primeira parte, que corresponde às primeiras cinco novas linhas, corresponde à declaração de variáveis. primeiro o tipo que pretendemos e depois o nome da variável.

De seguida, o código disponibiliza o resultado das operações básicas de cálculo com as variáveis declaradas.

Aproveitando a declaração das variáveis e fazendo a validação de dados, foi utilizado o tratamento de excepções. Para isso, o C# utiliza o `try { } catch { }`. Desta forma, mesmo que o utilizador insira texto em vez de um valor inteiro, o programa não pára a sua execução.

1.3.7 OPERADORES

1.3.7.1 OPERADORES ARITMÉTICOS

Os operadores em C# são comuns à maior parte das linguagens. Vejamos a seguinte lista:

- Operador de adição: + (sinal positivo).
- Operador de subtracção: - (sinal negativo).
- Operador de divisão: / (barra).
- Operador de multiplicação: * (asterisco).
- Operador de resto da divisão inteira: % (sinal de percentagem).

Além destes operadores, o C# também utiliza o incremento (++) e o descremento (--).

Incremento

Quando surge a operação de incremento como sufixo, tipo `a++`, significa que a variável a irá ser incrementada em uma unidade.

Decremento

Quando surge a operação de decremento como sufixo, tipo `a--`, significa que a variável a irá ser decrementada em uma unidade.

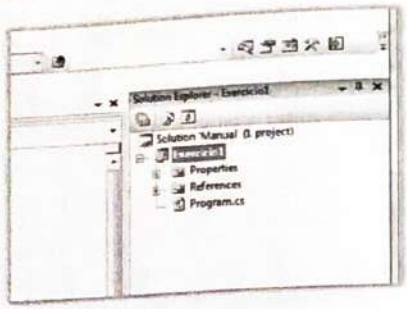
Estes (incremento e decremento) podem ser usados como prefixo (`++a`) ou sufixo (`a++`).

Prefixo – consideremos o seguinte exemplo:

```
int j;
int i=10;
j=++i;
```

Neste caso, a variável `j` recebe o valor de `I` depois de este ser incrementado, por isso ambas as variáveis (`i` e `j`) ficam com o valor de `11`.

da solução e uma solução pode ter vários projectos, neste caso o primeiro tem o mesmo nome da solução ("manual") e vamos alterar para "exercicio01". Para isso clicamos no nome do projecto e escolhemos a opção "rename" com o botão direito do rato. Este será o resultado:



Na janela temos a área de trabalho de programação.

1.3.5 O PRIMEIRO PROGRAMA

Analisemos com pormenor o conteúdo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Exercicio01
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

No código temos desde logo definidas as livrarias que o C# utiliza (estão precedidas da palavra *using*). Sempre que necessário poderemos acrescentar mais referências como por exemplo `using Linq`, se quisermos usar funções Linq de conexão a objectos SQL.

1.3.5.1 *NAMESPACE*

De seguida identificamos o *namespace*. Mas o que é um *namespace*?

Tudo em C# está declarado em *namespaces*. Um *namespace* é uma colecção de classes de definição de objectos da nossa programação. O facto de estarmos a declarar o namespace `Exercicio01` quer dizer que todas as classes que representarmos correspondem a este *namespace*. Recuando um pouco no manual, no primeiro capítulo, tínhamos dito que tudo em C# são classes e aqui está a evidência.

1.3.5.2 MAIN ( )

Na declaração da função `Main` temos `static void Main(string[] args)` onde `static void` indica que a função não retorna valores, senão seria `static int` para devolver um inteiro. Depois temos a possibilidade de referir que o executável do programa ("exercicio01.exe" que será o resultado da compilação do código). Podemos usar argumentos na execução do ficheiro executável separando os mesmos com espaços: `exercicio01.exe arg1 arg2 arg3`. Para usar esses argumentos programa bastará usar `args[i]` onde `i` representa a posição do argumento pretendida do *array*.

Vamos então agora desenvolver o "exercicio01". Com o exemplo simples de impressão de valores para o ecrã, iremos ver como podemos criar um projecto, em modo consola, em C#.

1.3.5.3 CÓDIGO C# O PRIMEIRO PROGRAMA "HELLO WORLD".

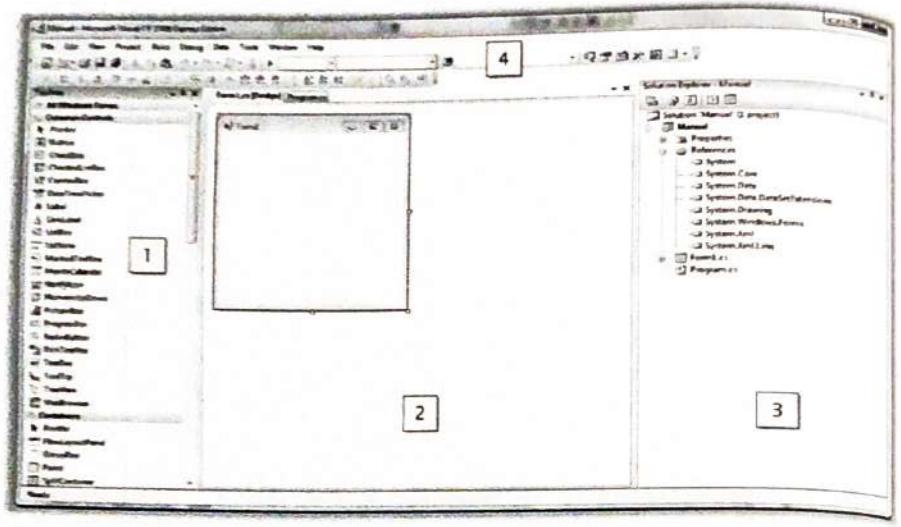
Desenvolva no ficheiro, o código identificado de seguida:

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

namespace Exercicio01
{
    class Program
    {
        static void Main(string[] args)
        {
        Console.WriteLine("Hello World!");
        Console.ReadKey();
        }
    }
}
```

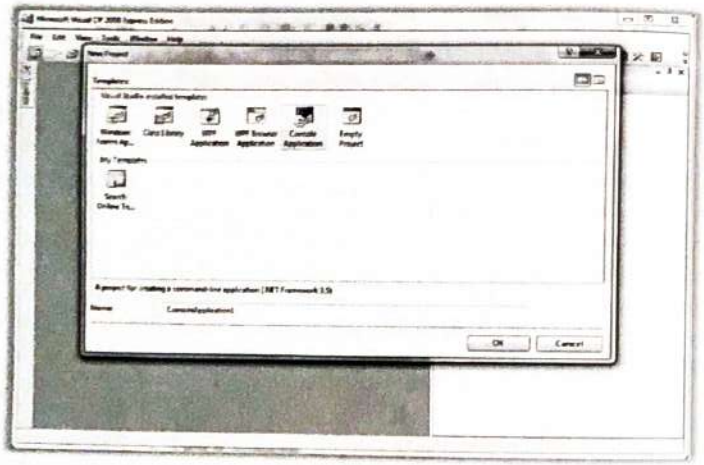
1.3.3 INTERFACE DE DESENVOLVIMENTO

Conheçamos a interface de desenvolvimento do Microsoft Visual C# 2008 Expres



1. Caixa de ferramentas
2. Área de trabalho
3. Explorador da solução
4. Barras de menus

Quando iniciamos um novo projecto em C#, termos de escolher umas das seguintes opções disponíveis no menu de *Create New Project*.



Poderemos recorrer aos modelos disponíveis e criar a aplicação:

- *Windows Forms Application*: aplicação em C# baseada em formulário Windows.
- *Class library*: definição de uma classe que origina ficheiros .dll a usar nas aplicações *Windows based*.
- *Windows Presentation Foundation client application* (.NET Framework 3.5).
- *Windows Presentation Foundation browser application* (.NET Framework 3.5).
- *Console Application*: criação de uma aplicação em modo *command-line* (.NET Framework 3.5).

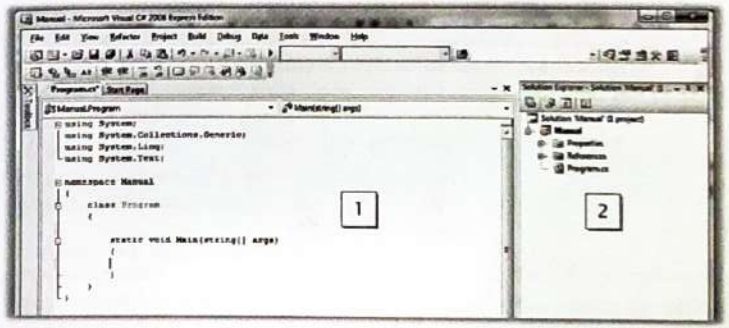
1.3.4 O PRIMEIRO PROJECTO

Para testarmos a linguagem e os tópicos principais de programação, iremos analisar um primeiro programa elaborado em modo consola.

Para isso vamos iniciar o Microsoft Visual C# Express Edition. Para obter o programa podem aceder directamente ao site da Microsoft e efectuar o download gratuitamente para a versão *Express*.

De seguida acedemos ao menu "File" → "New Project". No menu de Templates escolhemos "Console Application" e atribuímos o nome de "Manual".

Eis o nosso projecto disponível:



Na janela 1 podemos observar os ficheiros de programação "program.cs". Este ficheiro será o repositório do código em C#. Depois de apresentar a janela 2, iremos explicar o que a compõe. Podem observar que na janela 2 encontramos a "Solution Explorer" (área de exploração da solução). O primeiro item é o nome

criar um novo ficheiro de texto com três linhas simples. Vamos aproveitar e testar se o ficheiro existe, se sim mostra o conteúdo do mesmo, se não cria um novo e adiciona o texto pretendido.

Vejamos o seguinte código:

```
string path = Path.GetFullPath ("ficheiro.txt");
      FileInfo fi1 = new FileInfo(path);
      //testar se o ficheiro existe
      if (!fi1.Exists)
      {
          //Se o ficheiro não existe, cria de novo
          using (StreamWriter sw = fi1.CreateText())
          {
              //escreve o conteúdo para o ficheiro
              sw.WriteLine("Esta é a primeira linha de
código");
              sw.WriteLine("e");
              sw.WriteLine("Esta é a segunda linha de
código");
          }
      }
      //se o ficheiro existe, abre e mostra o seu
conteúdo
      using (StreamReader sr = fi1.OpenText())
      {
          string s = "";
          while ((s = sr.ReadLine()) != null)
          {
              Console.WriteLine(s);
          }
      }
Console.ReadKey();
```

Na primeira linha, a variável `path` fica com o caminho onde está a aplicação, como estamos a executar o *debug* da aplicação, este está na pasta de projecto\bin\debug\ficheiro.txt de seguida vamos criar um *alias* para o `fileinfo`. Se este não existir, cria um ficheiro novo e, com a `stream sw`, escreve três linhas de texto.

No final, abre o ficheiro e escreve o seu conteúdo, até que o texto obtido da leitura do ficheiro acabe.

Agora tentemos algo mais arrojado. O objectivo será copiar o ficheiro anterior para um novo. No fim de o copiar, eliminamos o mesmo. Portanto, não passa de um teste para testar as funcionalidades.

Observando o código seguinte, podemos desenvolvê-lo logo de seguida ao anterior:

```
try
      {
          string path2 = Path.GetTempFileName();
          FileInfo fi2 = new FileInfo(path2);

          //ter a certeza que o destino não existe e
proceder à
          //eliminação
              fi2.Delete();

          //copia o ficheiro 1 para o caminho de path2
              fi1.CopyTo(path2);
                Console.WriteLine("{0} foi copiado para
{1}.", path, path2);

              //elimina o ultimo ficheiro criado
              fi2.Delete();
                Console.WriteLine("{0} foi eliminado
com sucesso.", path2);

          }
      catch (Exception e)
          {
              Console.WriteLine("o proceso falhou:
{0}", e.ToString());
          }
       Console.ReadKey();
```

Como podemos observar, trabalhar com ficheiros de texto é relativamente fácil. Vejamos outro exercício de exemplo. O objectivo do "exercício10" será, criar um ficheiro de texto com determinado conteúdo e depois carregar-lo para uma `arraylist`, procurando determinado texto no seu conteúdo. Primeiro não devemos esquecer de adicionar o `using System.IO` para utilizar os ficheiros e também `using System.Collections` para usar a `arraylist`. Vamos então elaborar o código do `main()`.

Observem o exemplo:

```
StreamWriter sw = new
StreamWriter("ficheiro.txt");
for (int i = 0; i < 10; i++)
{
    sw.WriteLine("valor " + i);
}
sw.Close();
```

1.3.10 VECTORES (*ARRAYS*)

A versatilidade do C# na gestão de estruturas do tipo de vector, ou mais chamado de *array*, é realmente fantástica. Ao contrário do que encontramos em linguagem C em que os *arrays* podem ser de diversos tipos de dados, como `int`, `float` ou mesmo `struct`, o seu índice era sempre um inteiro. Em C# temos mais opções disponíveis. Comecemos por declarar *arrays*:

Declaração de *arrays*

```
int teste = 5;
int[] arraydeinteiros = new int[teste];
int[] outroarraydeinteiros = new int[5];
```

Neste exemplo específico estamos a declarar dois *arrays* de inteiros, ambos com cinco inteiros. Podemos também indicar o número de índices recorrendo a variáveis (vejam o exemplo `int[] arraydeinteiros = new int[teste];`), algo que não acontecia em linguagem C, por exemplo. No segundo exemplo, declaramos o *array* `outroarraydeinteiros` com cinco inteiros.

Depois de declarado o *array*, vamos inserir valores nos índices. Podemos recorrer simplesmente à posição e efectuar a atribuição.

Definir valores para o *array*

```
arraydeinteiros[0] = 15;
arraydeinteiros[1] = 25;
arraydeinteiros[2] = 15;
```

Como podemos observar, a atribuição pode ser efectuada directamente na posição pretendida ou então também poderíamos recorrer a uma variável que representa-se o índice (`arraydeinteiros[i] = 15;` em que `i` é uma variável inteira que percorre todo o *array*). Depois de inseridos os dados, resta-nos percorrer o mesmo e mostrar o seu conteúdo.

Mostrar valores para o *array*

```
for(int i=0;i<arraydeinteiros.Length;i++)
        Console.WriteLine(arraydeinteiros[i]);

//outra forma de percorrer um vector
foreach (int x in arraydeinteiros)
     Console.WriteLine(x);
```


A sintaxe do ciclo `for` é `for(valor inicial; limite; incremento/decremento)`. Neste exemplo, a função `Length` devolve o número de registos que tem o *array*.

Outra forma de percorrer o *array* é usando a função `foreach`, que será utilizada em vectores que não tenham índices inteiros mas outro tipo de índices.

Outro *array* que o C# permite utilizar é o `arraylist`. O `arraylist` está disponível no `namespace System.Collections`. Os `arraylist` são objectos que podem guardar, sobre a forma de um vector, qualquer tipo de valor. Vejamos o exemplo seguinte:

Definir um `arraylist`

```
ArrayList Inteiros = new ArrayList();
Inteiros.Add(15);
Inteiros.Add("teste");
Inteiros.Add(24);
//consultar o conteúdo do vector
foreach (object x in Inteiros)
        Console.WriteLine(x);
//outra forma de consultar o conteúdo do vector
for(int k=0; k<Inteiros.Count;k++)
        Console.WriteLine(Inteiros[k]);
```

Outro exemplo de aplicação são os *arrays* de *arrays*. Que funcionam como uma matriz em que podemos definir valores numa tabela e gerir da mesma forma:

Trabalhar com *array* de *arrays*

```
// Array de Array
int[][] arrayDeArray = new int[2][];
arrayDeArray[0] = new int[5];
arrayDeArray[1] = new int[2];
for (int i = 0; i < arrayDeArray.Length; i++)
     {
      for (int j = 0; j < arrayDeArray[i].Length; j++)
        {
        arrayDeArray[i][j] = i + j;
        }
     }
for (int i = 0; i < arrayDeArray.Length; i++)
     {
      for (int j = 0; j < arrayDeArray[i].Length; j++)
          {
        Console.WriteLine("{0},{1} : {2}", i, j,
      arrayDeArray[i][j]);
          }
     }
```

1.3.8 *BOXING* E *UNBOXING*

Em C# tudo são objectos. Uma das particularidades entre o tipo de dados e o possibilidade de efectuar a conversão directamente entre tipo e objecto e objecto e tipo. A este processo chamamos de *Boxing* e *unboxing* respectivamente.

Façamos o "exercício03". Na solução, criar-se um novo projecto em consola com o nome "exercicio03" (em caso de dúvida, consultar no texto anterior como foi criado o "exercicio02").

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

namespace Exercicio03
{
    class Program
    {
        static void Main(string[] args)
        {

         //Foi declarado o inteiro om o valor 100
                  int x = 100;

         //foi aplicado o "boxing"
            object obj = x;

         //foi aplicado o unboxing
            int j= (int)obj;
            Console.WriteLine("j:{0} / i:{1} / obj:{2}",
j, x,obj);
            Console.ReadKey();
        }
    }
}
```

Esta funcionalidade é muito útil quando, por exemplo, podemos gerar e gerir um *array* de objectos, sem que o seu conteúdo seja similar ou pelo menos do mesmo tipo.

A forma onde encontramos esta funcionalidade é na construção de aplicações visuais, baseadas em formulários, em que todos os objectos utilizados no form estão referenciados num *array* de objectos.

1.3.9 *CASTING*

O *casting* reflecte a possibilidade de converter os valores em vários formatos, como por exemplo um inteiro em duplo, como um duplo em inteiro. Esta funcionalidade já nos era familiar de outras linguagens de programação como a linguagem C.

Para testar a funcionalidade, criamos o "exercício03".

Vejamos o exemplo:

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

namespace Exercicio03
{
    class Program
    {
        static void Main(string[] args)
        {
            int myInt = 10;
            int otherInt = 100;
            double myDouble = myInt;
            myDouble = 200.482;
            otherInt =
Convert.ToInt32(myDouble);
            myInt = (int)myDouble;
            Console.WriteLine(" myInt: {0},
otherInt: {1}, myDouble: {2}", myInt,
otherInt, myDouble);
            Console.ReadKey();
        }
    }
}
```

Foram declarados dois inteiros (`myInt` e `otherInt`) e um duplo (`myDouble`). O primeiro exemplo de casting é feito na linha `otherInt = Convert.ToInt32(myDouble);` recorrendo à função `Convert`. Neste caso estamos a converter o `myDouble` (que é um duplo) num inteiro (`System.Int32`). Outra possibilidade de implementação é recorrendo ao outro exemplo `myInt = (int)myDouble;` em que convertemos o `myDouble` em inteiro.

O resultado final:

```
myInt:200, otherInt:200, myDouble: 200.482
```

Os Textos de Apoio ao Aluno, Técnicas e Linguagens de Programação 11, da componente Técnica, Tecnológica e Prática, destinam-se à seguinte área de formação e respectivos cursos:

ÁREA DE INFORMÁTICA

CURSOS
Técnico de Informática de Gestão
Técnico de Informática/Sistemas Multimédia

REPÚBLICA DE ANGOLA
MINISTÉRIO DA EDUCAÇÃO

REFORMA EDUCATIVA

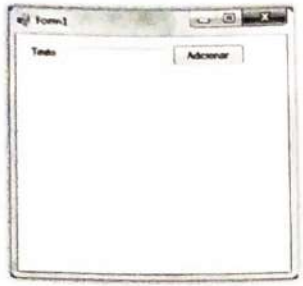
RETEP
REFORMA DO ENSINO
TÉCNICO-PROFISSIONAL

reditep
EDIÇÕES REDITEP

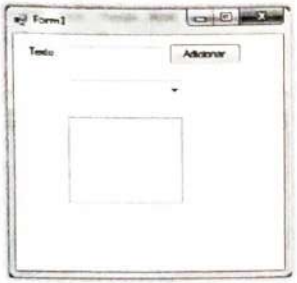
lusis

Começamos por colocar uma "label" e uma `textbox`. Vejamos o exemplo:

Depois colocamos um botão no formulário, a que lhe vamos dar o nome de "Adicionar".



E por fim, vamos criar uma `combobox` e uma `listbox`. Quando inserirmos texto na `textbox` e depois fizermos adicionar, a `combobox` ficará com mais opções para escolher e quando escolhermos uma opção da `combobox`, vamos fazer com que a selecção seja adicionada à `listBox`.



Para isso teremos que desenvolver duas opções, a primeira que quando fizermos adicionar, seja colocada a nova opção na "combo" e a segundo, quando a "combo" for actualizada.

Vejamos a primeira com detalhe, quando activarmos o botão "adicionar":

```
private void button1_Click(object sender, EventArgs e)
        {
            comboBox1.Items.Add(textBox1.Text);
            comboBox1.Refresh();
        }
```

No evento `button1_click()`, em que `button1` é o nome do nosso objecto botão e o evento será o "click", fazemos com que o texto introduzido na `textbox` seja adicionado à "combo" recorrendo ao comando `comboBox1.Items.Add(textBox1.Text);` em que a função `Items.add()` encarrega-se de adicionar um novo elemento à "combo". Depois optamos por efectuar um `refresh()` à "combo" para que esta actualize os seus elementos.

Na segunda tarefa, teremos de actualizar a `listbox` em função do item seleccionado na "combo". Para isso desenvolvemos o seguinte código:

```
private void comboBox1_SelectedIndexChanged(object
sender, EventArgs e)
  {
  listBox1.Items.Add(comboBox1.SelectedItem.ToString ());
  listBox1.Refresh();
  }
```

Como as opções, quer da "combo" quer da `listbox` são texto, convertemos para texto a opção seleccionada na "combo" e adicionamo-la à `listbox`, recorrendo à função `Item.add()` que tem um funcionamento idêntico ao da `combobox`.

Se executarmos o programa, primeiro inserimos texto na `textbox` e fazemos adicionar (para comprovar o funcionamento, faça este passo por diversas vezes, adicionando várias opções à "combo"). Depois bastará seleccionar um item da "combo" e ele aparecerá.

1.4.4 EDITOR DE TEXTO PERSONALIZADO

Nos sistemas operativos actuais existem aplicativos adicionais que nos permitem editar ficheiros de texto com grande facilidade. Um desses exemplos é o "bloco de notas", ou Notepad, sendo este o nome pelo qual é mais conhecido.

Assim, vamos desenvolver um novo exercício, o exercicio19. O desafio será criar um editor de texto, para `"ficheiros.txt"`, que permita abrir várias janelas em simultâneo (o que não acontece actualmente com o Notepad) e permitir editar vários ficheiros de cada vez.

1.3.14 ENUMERAÇÕES

As enumerações podem ser utilizadas para definir uma lista de constantes a utilizar no código. Por exemplo, podemos criar uma enumeração para os dias da semana e invocar o seu valor se necessário. A vantagem principal na utilização das enumerações é o facto de estas serem constantes e daí garantirem a fiabilidade dos dados durante e execução do programa.

Assim, as enumerações definem conjuntos de constantes simbólicas:

- Conversão de/para inteiro, mas só explícita.
- Operadores aplicáveis:
  - Comparação: `== > < >= <= !=`
  - Bit-a-bit: `& | ^ ~`
  - Outros: `+ - ++ -- sizeof`
  - Pode especificar-se o tipo subjacente (por omissão `int`).
  - `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` ou `ulong`.
  - Herdam implicitamente de `System.Enum`.

Exemplo:

```
enum Color: byte {
            Red = 1,
            Green = 2,
            Blue = 4,
            Black = 0,
            White = Red | Green | Blue
            }
      Color c = Color.Red;
```

Elaboremos um novo exercício com as enumerações. Assim, iremos criar o "exercicio08". No "exercicio08" vamos declarar, de forma global a todo o programa, isto é, deverão ser declaradas na classe `program` mas antes da função membro `main()`, a estrutura.

```
enum Cores { Verde = 1, Vermelho, Amarelo }
```

De seguida, na função `main()` iremos elaborar um programa capaz de perguntar ao utilizador a sua cor pretendida. Depois, com o auxílio do `switch`, vamos testar a opção introduzida e mostrar o resultado da escolha.

Então, vamos desenvolver o seguinte código:

```
Console.WriteLine("Escolha a cor: ");
Console.WriteLine("1 - Verde");
Console.WriteLine("2 - Vermelho");
Console.WriteLine("3 - Amarelo");
int op = int.Parse(Console.ReadLine());
switch (op)
{
    case (int)Cores.Verde:
        Console.WriteLine("Escolheu Verde");
        break;
    case (int)Cores.Vermelho:
        Console.WriteLine("Escolheu Vermelho");
        break;
    case (int)Cores.Amarelo:
        Console.WriteLine("Escolheu Amarelo");
        break;
}
Console.ReadKey();
```

Passemos à interpretação do bloco de código. No início, declaramos uma enumeração `enum Cores { Verde = 1, Vermelho, Amarelo }`, com os valores `Verde = 1, Vermelho, Amarelo`. No caso do valor `Verde`, colocamos uma regra sendo este o primeiro valor. O que o `C#` irá interpretar é que o `Verde` é o primeiro valor e posso chamar os outros directamente pelo valor ou então recorrer ao índice. Em seguida, construímos um "menu" em que o utilizador irá escolher uma das opções. Com a opção guardada em `op` vamos fazer o `switch`. Uma potencialidade do uso de enumerações tem a ver com o facto de pudermos usar o índice, convertendo simplesmente fazendo um *cast*, ou então podia chamar simplesmente o valor da enumeração.

O resultado final do código será a impressão na consola da cor a que corresponde o valor do índice inserido.

1.3.15 FICHEIROS EM MODO CONSOLA

O tema dos ficheiros em modo consola não desperta muito interesse, no entanto é uma excelente prática se depois os usarmos para fazer *debug* ou *log* de acessos às aplicações ou mesmo perceber como trabalhar com os ficheiros de forma a mais tarde exportar e importar informações para formatos compatíveis ou mesmo construir ficheiros como XML, embora neste caso específico o `C#` tem como o fazer de forma mais directa e sem qualquer esforço.

Vamos desenvolver um novo exercício, o "exercicio09". Para isso e para que o `C#` importe as livrarias que precisamos, termos de adicionar `using System.IO;`. Depois iremos desenvolver um pequeno código que permita