Unidade 2

2 Orientação a objetos básica

Ao término deste capítulo, você será capaz de:

- Dizer o que é e para que serve orientação a objetos;
- Conceituar classes, atributos e comportamentos;
- Entender o significado de variáveis e objetos na memória.

2.1 - Motivação: problemas do paradigma procedural

Orientação a objetos é uma maneira de programar que ajuda na organização e resolve muitos problemas enfrentados pela programação procedural.

Consideremos o clássico problema da validação de um CPF. Normalmente, temos um formulário, no qual recebemos essa informação, e depois temos que enviar esses caracteres para uma função que irá validá-lo, como no pseudo código abaixo:

```
cpf = formulario->campo_cpf
valida(cpf)
```

Alguém te obriga a sempre validar esse CPF? Você pode, inúmeras vezes, esquecer de chamar esse validador.

Mais: considere que você tem 50 formulários e precise validar em todos eles o CPF. Se sua equipe tem 3 programadores trabalhando nesses formulários, quem fica responsável por essa validação? Todos!

A situação pode piorar: na entrada de um novo desenvolvedor, precisaríamos avisá-lo que sempre devemos validar o cpf de um formulário. É nesse momento que nascem aqueles guias de programação para o desenvolvedor que for entrar nesse projeto - às vezes é um documento enorme. Em outras palavras, **todo** desenvolvedor precisa ficar sabendo de uma quantidade enorme de informações, que, na maioria das vezes, não está realmente relacionado à sua parte no sistema, mas ele **precisa** ler tudo isso, resultando um entrave muito grande!

Outra situação onde ficam claros os problemas da programação procedural, é quando nos encontramos na necessidade de ler o código que foi escrito por outro desenvolvedor e descobrir como ele funciona internamente.

Um sistema bem encapsulado não deveria gerar essa necessidade. Em um sistema grande, simplesmente não temos tempo de ler uma parte grande do código.

Considerando que você não erre aí e que sua equipe tenha uma comunicação muito boa (perceba que comunicação excessiva pode ser prejudicial e atrapalhar o andamento), ainda temos outro problema: imagine que, agora, em todo formulário, você também quer que a idade do cliente seja validada - precisa ser maior de 18 anos. Vamos ter de colocar um if... mas onde? Espalhado por todo seu código... Mesmo que se crie outra função para validar, precisaremos incluir isso nos nossos 50 formulários já existentes. Qual é a chance de esquecermos em um deles? É muito grande. A responsabilidade de verificar se o cliente tem ou não tem 18 anos, ficou espalhada por todo o seu código.

Seria interessante poder concentrar essa responsabilidade em um lugar só, para não ter chances de esquecer isso.

Melhor ainda seria se conseguíssemos mudar essa validação e os outros programadores nem precisassem ficar sabendo disso. Em outras palavras, eles criariam formulários e um único programador seria responsável pela validação: os outros nem sabem da existência desse trecho de código. Um mundo ideal? Não, o paradigma da orientação a objetos facilita tudo isso.

O problema é que não existe uma conexão entre seus dados! Não existe uma conexão entre seus dados e suas funcionalidades! A idéia é ter essa amarra através da linguagem.

Quais as vantagens?

Orientação a objetos vai te ajudar em muito em se organizar e escrever menos, além de concentrar as responsabilidades nos pontos certos, flexibilizando sua aplicação, **encapsulando** a lógica de negócios.

Outra enorme vantagem, onde você realmente vai economizar montanhas de código, é o **polimorfismo das referências**, que veremos em um posterior capítulo.

2.2 - Criando um tipo

Considere um programa para um banco, é bem fácil perceber que uma entidade extremamente importante para o nosso sistema é a conta. Nossa idéia aqui é generalizarmos alguma informação, juntamente com funcionalidades que toda conta deve ter.

O que toda conta tem e é importante para nós?

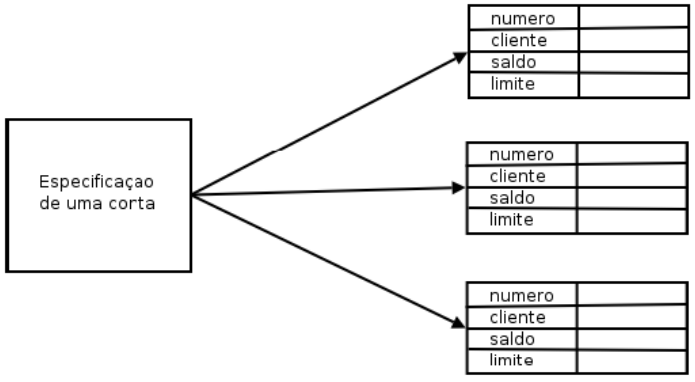
- Número da conta
- Nome do cliente
- Saldo
- Limite

O que toda conta faz e é importante para nós? Isto é, o que gostaríamos de "pedir à conta".

- Saca uma quantidade x
- Deposita uma quantidade x
- Imprime o nome do dono da conta
- Devolve o saldo atual
- Transfere uma quantidade x para uma outra conta y
- Devolve o tipo de conta

Com isso, temos o projeto de uma conta bancária. Podemos pegar esse projeto e acessar seu saldo? Não.

O que temos ainda é o **projeto**. Antes, precisamos **construir** uma conta, para poder acessar o que ela tem, e pedir a ela que faça alguma coisa.



Repare na figura: apesar do papel do lado esquerdo especificar uma Conta, essa especificação é uma Conta? Nós depositamos e sacamos dinheiro desse papel? Não. Utilizamos a especificação da Conta para poder criar instâncias que realmente são contas, onde podemos realizar as operações que criamos.

Apesar de declararmos que toda conta tem um saldo, um número e uma agência no pedaço de papel (como à esquerda na figura), são nas instâncias desse projeto que realmente há espaço para armazenar esses valores.

Ao projeto da conta, isto é, a definição da conta, damos o nome de **classe**. Ao que podemos construir a partir desse projeto, as contas de verdade, damos o nome de **objetos**.

A palavra **classe** vem da taxonomia da biologia. Todos os seres vivos de uma mesma **classe** biológica têm uma série de **atributos** e **comportamentos** em comum, mas não são iguais, podem variar nos valores desses **atributos** e como realizam esses **comportamentos**.

Homo Sapiens define um grupo de seres que possuem características em comum, porém a definição (a idéia, o conceito) de um **Homo Sapiens** é um ser humano? Não. Tudo está especificado na **classe** Homo Sapiens, mas se quisermos mandar algúem correr, comer, pular, precisaremos de uma instância de **Homo Sapiens**, ou então de um **objeto** do tipo **Homo Sapiens**.

Um outro exemplo: uma receita de bolo. A pergunta é certeira: você come uma receita de bolo? Não.

Precisamos **instaciá-la**, criar um **objeto** bolo a partir dessa especificação (a classe) para utilizá-la. Podemos criar centenas de bolos a partir dessa classe (a receita, no caso), eles podem ser bem semelhantes, alguns até idênticos, mas são **objetos** diferentes.

Podemos fazer milhares de analogias semelhantes. A planta de uma casa é uma casa? Definitivamente não. Não podemos morar dentro da planta de uma casa, nem podemos abrir sua porta ou pintar suas paredes.

Precisamos, antes, construir instâncias a partir dessa planta. Essas instâncias, sim, podemos pintar, decorar ou morar dentro.

Pode parecer óbvio, mas a dificuldade inicial do paradigma da orientação a objetos é justo saber distinguir o que é classe e o que é objeto. É comum o iniciante utilizar, obviamente de forma errada, essas duas palavras como sinônimos.

2.3 - Uma classe em C#

Vamos começar apenas com o que uma Conta tem, e não com o que ela faz (veremos logo em seguida).

Um tipo desses, como o especificado de Conta acima, pode ser facilmente traduzido para C#:

```
public class Conta
{
	int numero;
	String nome;
	double saldo;
	double limite;
	  // ..
}
```

String

String é uma classe em C#. Ela guarda uma palavra, isso é um punhado de caracteres. Como estamos aprendendo o que é uma classe, entenderemos com detalhes a classe String apenas em capítulos posteriores.

Por enquanto, declaramos o que toda conta deve ter. Estes são os **atributos** que toda conta, quando criada, vai ter. Repare que essas variáveis foram declaradas fora de um bloco, diferente do que fazíamos quando tinha aquele Main. Quando uma variável é declarada diretamente dentro do escopo da classe, é chamada de variável de objeto, ou atributo.

2.4 - Criando e usando um objeto

Agora, temos uma classe em C# que especifica o que todo objeto dessa classe deve ter. Mas como usá-la? Além dessa classe, ainda teremos o **Program.cs** e a partir dele é que iremos utilizar a classe Conta.

Para criar (construir, instanciar) uma Conta, basta usar a palavra chave new, utilizamos também os parênteses, que descobriremos o que são, exatamente, em um capítulo posterior:

```
public class Program
{
	static void Main(String[] args)
	{
		new Conta();
	}
}
```

Bem, o código acima cria um objeto do tipo Conta, mas como acessar esse objeto que foi criado? Precisamos ter alguma forma de nos referenciarmos a esse objeto. Precisamos de uma variável:

```
public class Program
{
	static void Main(String[] args)
	{
		Conta minhaConta = null;
		minhaConta = new Conta();
	}
}
```

Pode parecer estranho escrevermos duas vezes Conta: uma vez na declaração da variável e outra vez no uso do new. Mas há um motivo, que entenderemos também posteriormente.

Através da variável minhaConta, agora, podemos acessar o objeto recém criado para alterar seu nome, seu saldo etc:

```
public class Programa
{
	static void Main(String[] args)
	{
		Conta minhaConta;
		minhaConta = new Conta();

		minhaConta.nome = "Duke";
		minhaConta.saldo = 1000.0;

		Console.WriteLine("Saldo atual: " + minhaConta.saldo);
	}
}
```

É importante fixar que o ponto foi utilizado para acessar algo em minhaConta. Agora, minhaConta pertence ao Duke, e tem saldo de mil reais.

2.5 – Métodos

Dentro da classe, também declararemos o que cada conta faz e como isto é feito - os comportamentos que cada classe tem, isto é, o que ela faz. Por exemplo, de que maneira que uma Conta saca dinheiro?

Especificaremos isso dentro da própria classe Conta, e não em um local desatrelado das informações da própria Conta. É por isso que essas "funções" são chamadas de **métodos**. Pois é a maneira de fazer uma operação com um objeto.

Queremos criar um método que **saca** uma determinada **quantidade** e não devolve **nenhuma informação** para quem acionar esse método:

```
public class Conta
{
      int numero;
      String nome;
      double saldo;
      double limite;
        // ..

      public void Saca(double quantidade)
      {
            double novoSaldo = this.saldo - quantidade;
            this.saldo = novoSaldo;
      }
}
```

A palavra chave void diz que, quando você pedir para a conta sacar uma quantia, nenhuma informação será enviada de volta a quem pediu.

Quando alguém pedir para sacar, ele também vai dizer quanto quer sacar. Por isso precisamos declarar o método com algo dentro dos parênteses - o que vai aí dentro é chamado de **argumento** do método (ou **parâmetro**). Essa variável é uma variável comum, chamada também de temporária ou local, pois, ao final da execução desse método, ela deixa de existir.

Dentro do método, estamos declarando uma nova variável. Essa variável, assim como o argumento, vai morrer no fim do método, pois este é seu escopo. No momento que vamos acessar nosso atributo, usamos a palavra chave this para mostrar que esse é um atributo, e não uma simples variável. (veremos depois que é opcional)

Repare que, nesse caso, a conta pode estourar o limite fixado pelo banco. Mais para frente, evitaremos essa situação, e de uma maneira muito elegante.

Da mesma forma, temos o método para depositar alguma quantia:

```
public void Deposita(double quantidade)
{
      this.saldo += quantidade;
}
```

Observe que, agora, não usamos uma variável auxiliar e, além disso, usamos a abreviação += para deixar o método bem simples. O += soma quantidade ao valor antigo do saldo e guarda no próprio saldo, o valor resultante.

Para mandar uma mensagem ao objeto e pedir que ele execute um método, também usamos o ponto. O termo usado para isso é **invocação de método**.

O código a seguir saca dinheiro e depois deposita outra quantia na nossa conta:

```
public class SacaEDeposita
{
	static void Main(String[] args)
	{
		// criando a conta
		Conta minhaConta;
		minhaConta = new Conta();

		// alterando os valores de minhaConta
		minhaConta.nome = "Duke";
		minhaConta.saldo = 1000;

		// saca 200 reais
		minhaConta.Saca(200);

		// deposita 500 reais
		minhaConta.Deposita(500);
		Console.WriteLine(minhaConta.saldo);
	}
}
```

Uma vez que seu saldo inicial é 1000 reais, se sacarmos 200 reais, depositarmos 500 reais e imprimirmos o valor do saldo, o que será impresso?

2.6 - Métodos com retorno

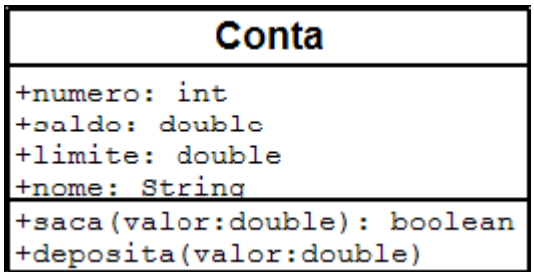
Um método sempre tem que retornar alguma coisa, nem que essa coisa seja nada, como nos exemplos anteriores onde estávamos usando o void.

Um método pode retornar um valor para o código que o chamou. No caso do nosso método saca podemos devolver um valor booleano indicando se a operação foi bem sucedida.

```
public class Conta
{
	// ... outros metodos e atributos ...
	public bool Saca(double valor)
	{
		if (this.saldo < valor)
		{
		    return false;
		}
		else
		{
		    this.saldo = this.saldo - valor;
		    return true;
		}
	}
}
```

Agora a declaração do método mudou! O método saca não tem void na frente, isto quer dizer que, quando é acessado, ele devolve algum tipo de informação. No caso,

um bool. A palavra chave return indica que o método vai terminar ali, retornando tal informação.



Exemplo de uso:

```
minhaConta.saldo = 1000;
bool consegui = minhaConta.Saca(2000);
if (consegui)
{
	Console.WriteLine("Consegui sacar");
}
else
{
	Console.WriteLine("Não consegui sacar");
}
```

Ou então, posso eliminar a variável temporária, se desejado:

```
minhaConta.saldo = 1000;
Console.WriteLine(minhaConta.Saca(2000));
```

Mais adiante, veremos que algumas vezes é mais interessante lançar uma exceção (exception) nesses casos.

Meu programa pode manter na memória não apenas uma conta, como mais de uma:

```
public class TestaDuasContas
{
	static void Main(String[] args)
	{
		// criando a conta
		Conta minhaConta;
		minhaConta = new Conta();
		minhaConta.saldo = 1000;

		Conta meuSonho;
		meuSonho = new Conta();
		meuSonho.saldo = 1500000;
	}
}
```

2.7 - Objetos são acessados por referências

Quando declaramos uma variável para associar a um objeto, na verdade, essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de **referência**.

É por esse motivo que, diferente dos tipos primitivos como int e long, precisamos dar new depois de declarada a variável:

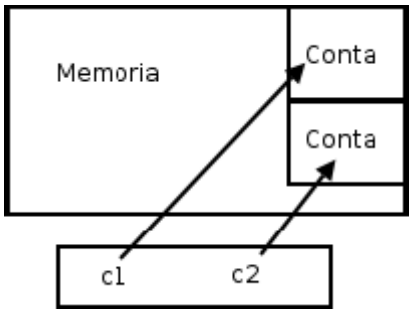
```
static void Main(String[] args)
{
      Conta c1;
      c1 = new Conta();

      Conta c2;
      c2 = new Conta();

}
```

O correto aqui, é dizer que c1 se refere a um objeto. **Não é correto** dizer que c1 é um objeto, pois c1 é uma variável referência, apesar de, depois de um tempo, os programadores C# falem "Tenho um **objeto c** do tipo **Conta**", mas apenas para encurtar a frase "Tenho uma **referência c** a um **objeto** do tipo **Conta**".

Basta lembrar que, em C#, **uma variável nunca é um objeto**. Todo objeto em C#, sem exceção, é acessado por uma variável referência.

Esse código nos deixa na seguinte situação:

```
Conta c1;
c1 = new Conta();

Conta c2;
c2 = new Conta();
```



Internamente, c1 e c2 vão guardar um número que identifica em que posição da memória aquela Conta se encontra. Dessa maneira, ao utilizarmos o "." para navegar, o C# vai acessar a Conta que se encontra naquela posição de memória, e não uma outra.

Para quem conhece, é parecido com um ponteiro, porém você não pode manipulá-lo e utilizá-lo para guardar outras coisas.

Agora vamos a um outro exemplo:

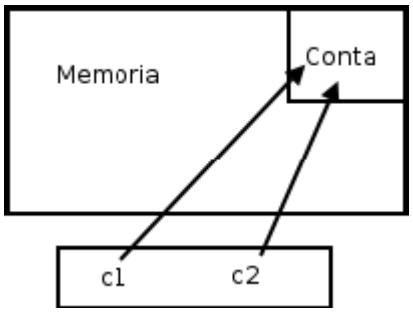
```
public class TestaReferencias
{
	static void Main(String[] args)
	{
		Conta c1 = new Conta();
		c1.Deposita(100);

		Conta c2 = c1;// linha importante
		c2.Deposita(200);


		Console.WriteLine(c1.saldo);
		Console.WriteLine(c2.saldo);
	}
}
```

Qual é o resultado do código acima? O que aparece ao rodar?

O que acontece aqui? O operador = copia o valor de uma variável. Mas qual é o valor da variável c1? É o objeto? Não. Na verdade, o valor guardado é a referência (**endereço**) de onde o objeto se encontra na memória principal.

Na memória, o que acontece nesse caso:

```
Conta c1 = new Conta();
c1.Deposita(100);

Conta c2 = c1;
```



Quando fizemos c2 = c1, c2 passa a fazer referência para o mesmo objeto que c1 referencia nesse instante.

Então, nesse código em específico, quando utilizamos c1 ou c2 estamos nos referindo exatamente ao **mesmo** objeto! Elas são duas referências distintas, porém apontam para o **mesmo** objeto! Compará-las com "==" irá nos retornar true, pois o valor que elas carregam é o mesmo!

Outra forma de perceber, é que demos apenas um new, então só pode haver um objeto Conta na memória.

Atenção: não estamos discutindo aqui a utilidade de fazer uma referência apontar pro mesmo objeto que outra. Essa utilidade ficará mais clara quando passarmos variáveis do tipo referência como argumento para métodos.

new

O que exatamente faz o new?

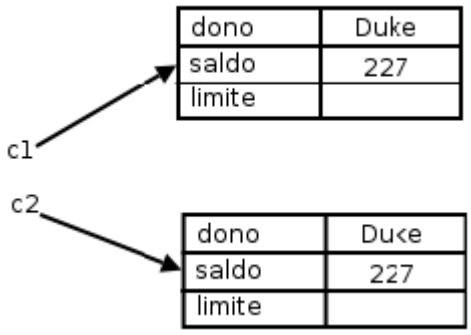
O new executa uma série de tarefas, que veremos mais adiante.

Mas, para melhor entender as referências no C#, saiba que o new, depois de alocar a memória para esse objeto, devolve uma "flecha", isto é, um valor de referência. Quando você atribui isso a uma variável, essa variável passa a se referir para esse mesmo objeto.

Podemos ver outra situação:

```
public class TestaIgualdade
{
      static void Main(String[] args)
      {
            Conta c1 = new Conta();
            c1.nome = "Duke";
            c1.saldo = 227;

            Conta c2 = new Conta();
            c2.nome = "Duke";
            c2.saldo  =227 ;


            if (c1 == c2)
            {
                Console.WriteLine("Contas iguais!");
            }
      }
}
```

O operador == compara o conteúdo das variáveis, mas essas variáveis não guardam o objeto, e sim o endereço em que ele se encontra. Como em cada uma dessas variáveis guardamos duas contas criadas diferentemente, eles estão em espaços diferentes da memória, o que faz o teste no if valer false. As contas podem ser equivalentes no nosso critério de igualdade, porém elas não são o mesmo objeto. Quando se trata de objetos, pode ficar mais fácil pensar que o == compara se os objetos (referências, na verdade) são o mesmo, e não se são iguais.



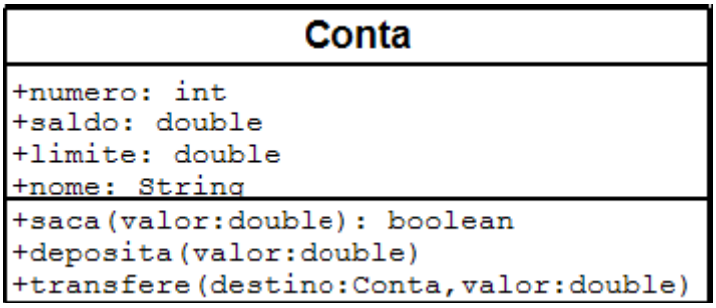
Para saber se dois objetos têm o mesmo conteúdo, você precisa comparar atributo por atributo. Veremos uma solução mais elegante para isso também.

2.8 - O método Transfere()

E se quisermos ter um método que transfere dinheiro entre duas contas? Podemos ficar tentados a criar um método que recebe dois parâmetros: conta1 e conta2 do tipo Conta. Mas cuidado: assim estamos pensando de maneira procedural.

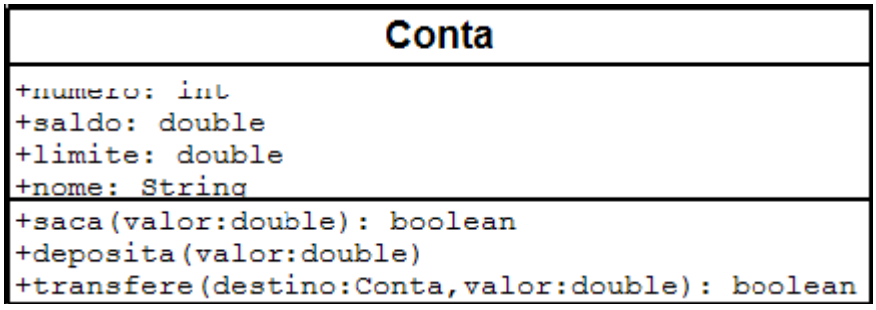
A idéia é que, quando chamarmos o método transfere, já teremos um objeto do tipo Conta (o this), portanto o método recebe apenas **um** parâmetro do tipo Conta, a Conta destino (além do valor):

```
public class Conta
{
        int numero;
        public string nome;
        public double saldo;
        double limite;

        public void Deposita(double quantidade)
        {
            this.saldo += quantidade;
        }
        public bool Saca(double valor)
        {
            if (this.saldo < valor)
            {
                return false;
            }
            else
            {
                this.saldo = this.saldo - valor;
                return true;
            }
        }
        public void Transfere(Conta destino, double valor)
        {
            this.saldo = this.saldo - valor;
            destino.saldo = destino.saldo + valor;
        }

}
```



Para deixar o código mais robusto, poderíamos verificar se a conta possui a quantidade a ser transferida disponível. Para ficar ainda mais interessante, você pode chamar os métodos deposita e saca já existentes para fazer essa tarefa:

```
public class Conta
{

	//atributos e métodos

	public bool Transfere(Conta destino, double valor)
	{
		bool retirou = this.Saca(valor);
		if (retirou == false)
		{
		    // não deu pra sacar!
		    return false;
		}
		else
		{
		    destino.Deposita(valor);
		    return true;
		}
	}

}
```



Quando passamos uma Conta como argumento, o que será que acontece na memória? Será que o objeto é clonado?

No C#, a passagem de parâmetro funciona como uma simples atribuição como no uso do "=". Então, esse parâmetro vai copiar o valor da variável do tipo Conta que for passado como argumento. E qual é o valor de uma variável dessas? Seu valor é um endereço, uma referência, nunca um objeto. Por isso não há cópia de objetos aqui.

Esse último código poderia ser escrito com uma sintaxe muito mais sucinta. Como?

Transfere para

Perceba que o nome deste método poderia ser TransferePara ao invés de só transfere. A chamada do método fica muito mais natural, é possível ler a frase em português que ela tem um sentido:

conta1.TransferePara(conta2, 50);

A leitura deste código seria "Conta1 transfere para conta2 50 reais".

2.9 - Continuando com atributos

As variáveis do tipo atributo, diferentemente das variáveis temporárias (declaradas dentro de um método), recebem um valor padrão. No caso numérico, valem 0, no caso de boolean, valem false.

Você também pode dar **valores default**, como segue:

```
public class Conta
{
	int numero = 1234;
	String dono = "Duke";
	String cpf = "123.456.789-10";
	double saldo = 1000;
	double limite = 1000;
}
```

Nesse caso, quando você criar um carro, seus atributos já estão "populados" com esses valores colocados.

Imagine que, agora, começamos a aumentar nossa classe Conta e adicionar nome, sobrenome e cpf do cliente dono da conta. Começaríamos a ter muitos atributos... e, se você pensar direito, uma Conta não tem nome, nem sobrenome nem cpf, quem tem esses atributos é um Cliente. Então podemos criar uma nova classe e fazer uma composição

Seus atributos também podem ser referências para outras classes. Suponha a seguinte classe Cliente:

```
public class Conta
{
	int numero = 1234;
	double saldo = 1000;
	double limite = 1000;
	Cliente titular;
}
```

```
public class Cliente
{
	String nome;
	String sobrenome;
	String cpf;
}
```

E dentro do Main da classe de teste:

```
static void Main(String[] args)
{
	Conta minhaConta = new Conta();
	Cliente c = new Cliente();
	minhaConta.titular = c;
	// ...
}
```

Aqui, simplesmente houve uma atribuição. O valor da variável c é copiado para o atributo titular do objeto ao qual minhaConta se refere. Em outras palavras, minhaConta agora tem uma referência ao mesmo Cliente que c se refere, e pode ser acessado através de minhaConta.titular.

Você pode realmente navegar sobre toda essa estrutura de informação, sempre usando o ponto:

```
Cliente clienteDaMinhaConta = minhaConta.titular;
clienteDaMinhaConta.nome = "Duke";
```

Ou ainda, pode fazer isso de uma forma mais direta e até mais elegante:

```
minhaConta.titular.nome = "Duke";
```

Um sistema orientado a objetos é um grande conjunto de classes que vai se comunicar, delegando responsabilidades para quem for mais apto a realizar determinada tarefa. A classe Banco usa a classe Conta que usa a classe Cliente, que usa a classe Endereco. Dizemos que esses objetos colaboram, trocando mensagens entre si. Por isso acabamos tendo muitas classes em nosso sistema, e elas costumam ter um tamanho relativamente curto.

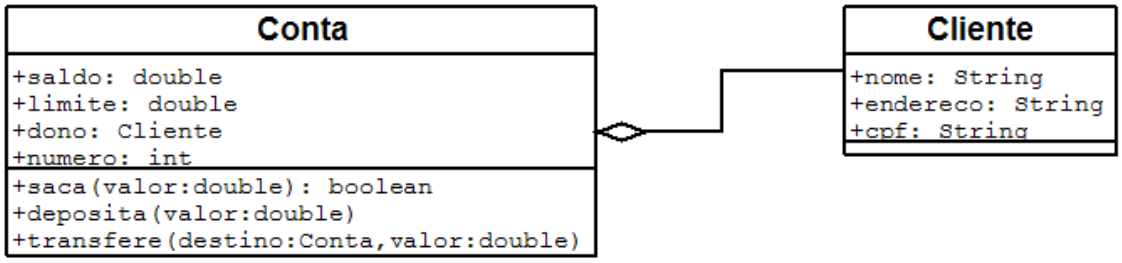
Mas, e se dentro do meu código eu não desse new em Cliente e tentasse acessá-lo diretamente?

```
public class Teste
{
	static void Main(String[] args)
	{
		Conta minhaConta = new Conta();
		Cliente c = new Cliente();
		minhaConta.titular = c;
		minhaConta.titular.nome = "Alexandre Rech";
		// ...

	}
}
```

Quando damos new em um objeto, ele o inicializa com seus valores default, 0 para números, false para boolean e null para referências. null é uma palavra chave em C#, que indica uma referência para nenhum objeto.

Se, em algum caso, você tentar acessar um atributo ou método de alguém que está se referenciando para null, você receberá um erro durante a execução (NullReferenceException, que veremos mais à frente). Da para perceber, então, que o new não traz um efeito cascata, a menos que você dê um valor default (ou use construtores, que também veremos mais a frente):

```
public class Conta
{
	public int numero;
	public double saldo;
	public double limite;
	public Cliente titular = new Cliente();

	//métodos...
}
```



Com esse código, toda nova Conta criada já terá um novo Cliente associado, sem necessidade de instanciá-lo logo em seguida da instanciação de uma Conta. Qual alternativa você deve usar? Depende do caso: para toda nova Conta você precisa de um novo Cliente? É essa pergunta que deve ser respondida.

Nesse nosso caso a resposta é não, mas depende do nosso problema.

Atenção: para quem não está acostumado com referências, pode ser bastante confuso pensar sempre em como os objetos estão na memória para poder tirar as conclusões de o que ocorrerá ao executar determinado código, por mais simples que ele seja. Com tempo, você adquire a habilidade de rapidamente saber o efeito de atrelar as referências, sem ter de gastar muito tempo para isso. É importante, nesse começo, você estar sempre pensando no estado da memória. E realmente lembrar que, no C# "uma variável nunca carrega um objeto, e sim uma referência para ele" facilita muito.

2.10 - Para saber mais: Uma Fábrica de Carros

Além do Banco que estamos criando, vamos ver como ficariam certas classes relacionadas a uma fábrica de carros. Vamos criar uma classe Carro, com certos atributos, que descrevem suas características, e com certos métodos, que descrevem seu comportamento.

```
public class Carro
{
	String cor;
	String modelo;
	double velocidadeAtual;
	double velocidadeMaxima;

	  //liga o carro
	public void Liga()
	{
		Console.WriteLine("O carro está ligado");
	}

	  //acelera uma certa quantidade
	public void Acelera(double quantidade)
	{
		double velocidadeNova = this.velocidadeAtual + quantidade;
		this.velocidadeAtual = velocidadeNova;
	}

	//devolve a marcha do carro
	public int PegaMarcha()
	{
		if (velocidadeAtual < 0)
		{
		    return -1;
		}
		if (velocidadeAtual >= 0 && this.velocidadeAtual < 40)
		{
		    return 1;
		}
		if (velocidadeAtual >= 40 && this.velocidadeAtual < 50)
		{
		    return 2;
		}
		return 3;
	}
}
```

Agora, vamos testar nosso Carro em um programa de testes:

```
public class TestaCarro
{
	static void Main(String[] args)
	{
		Carro meuCarro;
		meuCarro = new Carro();
		meuCarro.cor = "Verde";
		meuCarro.modelo = "Fusca";
		meuCarro.velocidadeAtual = 0;
		meuCarro.velocidadeMaxima = 80;

		// liga o carro
		meuCarro.Liga();

		// acelera o carro
		meuCarro.Acelera(20);
		Console.WriteLine(meuCarro.velocidadeAtual);
	}
}
```

Nosso carro pode conter também um Motor:

```
public class Carro
{
	String cor;
	String modelo;
	double velocidadeAtual;
	double velocidadeMaxima;
	Motor motor;
}
```

Podemos, agora, criar diversos Carros e mexer com seus atributos e métodos, assim como fizemos no exemplo do Banco.

2.11 - Um pouco mais...

1) Quando declaramos uma classe, um método ou um atributo, podemos dar o nome que quisermos, seguindo uma regra. Por exemplo, o nome de um método não pode começar com um número. Pesquise sobre essas regras.

2) Como você pode ter reparado, sempre damos nomes às variáveis com letras minúsculas. É que existem **convenções de código**, dadas pela Microsoft, para facilitar a legibilidade do código entre programadores. Essa convenção é muito seguida. Pesquise sobre ela no http://msdn.com, procure por "code conventions".

3) É necessário usar a palavra chave this quando for acessar um atributo? Para que, então, utilizá-la?

4) O exercício a seguir pedirá para modelar um "funcionário". Existe um padrão para representar suas classes em diagramas, que é amplamente utilizado, chamado **UML**. Pesquise sobre ele.

2.12 - Exercícios: Orientação a Objetos

O modelo de funcionários a seguir será utilizado para os exercícios de alguns dos posteriores capítulos.

O objetivo aqui é criar um sistema para gerenciar os funcionários do Banco.

Os exercícios desse capítulo são extremamente importantes.

1) Modele um funcionário. Ele deve ter o nome do funcionário, o departamento onde trabalha, seu salário (double), a data de entrada no banco (String), seu RG (String) e um valor booleano que indique se o funcionário está na empresa no momento ou se já foi embora. Você deve criar alguns métodos de acordo com sua necessidade. Além deles, crie um método bonifica que aumenta o salario do funcionário de acordo com o parâmetro passado como argumento. Crie, também, um método demite, que não recebe parâmetro algum, só modifica o valor booleano indicando que o funcionário não trabalha mais aqui. A idéia aqui é apenas modelar, isto é, só identifique que informações são importantes e o que um funcionário faz. Desenhe no papel tudo o que um Funcionario tem e tudo que ele faz.

2) Transforme o modelo acima em uma classe C#. Teste-a, usando uma outra classe que tenha o Main. Você deve criar a classe do funcionário chamada Funcionario, e a classe de teste você pode nomear como quiser.

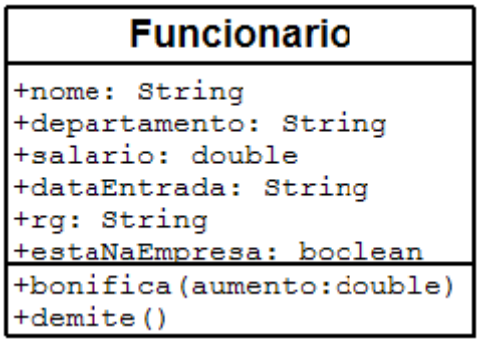
A de teste deve possuir o método Main.

Um esboço da classe:

```
public class Funcionario
{
        double salario;
        // seus outros atributos e métodos
        public void bonifica(double aumento)
        {
            // o que fazer aqui dentro?
        }
        public void demite()
        {
            // o que fazer aqui dentro?
        }
}
```

Você pode (e deve) compilar seu arquivo cs sem que você ainda tenha terminado sua classe Funcionario.

Isso evitará que você receba dezenas de erros de compilação de uma vez só. Crie a classe Funcionario, coloque seus atributos e, antes de colocar qualquer método, compile o arquivo cs. Não podemos "executá-la" pois não há um Main, mas assim verificamos que nossa classe Funcionario já está tomando forma.



Esse é um processo incremental. Procure desenvolver assim seus exercícios, para não descobrir só no fim do caminho que algo estava muito errado.

Um esboço da classe que possui o Main:

```
public class TestaFuncionario
{

        static void Main(String[] args)
        {
            Funcionario f1 = new Funcionario();

            f1.nome = "Fiodor";
            f1.salario = 100;
            f1.Bonifica(50);

            Console.WriteLine("salario atual:" + f1.salario);

      }
}
```

Incremente essa classe. Faça outros testes, imprima outros atributos e invoque os métodos que você criou a mais.

Lembre-se de seguir a convenção C#, isso é importantíssimo. Isto é, nomeDeAtributo, NomeDeMetodo, nomeDeVariavel, NomeDeClasse, etc...

Todas as classes no mesmo arquivo?

Por enquanto, você pode colocar todas as classes no mesmo arquivo e apenas compilar esse arquivo.

Porém, é boa prática criar um arquivo .cs para cada classe e, em determinados casos, você será obrigado a declarar uma classe em um arquivo separado, como veremos mais a frente. Isto não é importante para o aprendizado no momento.

3) Crie um método mostra(), que não recebe nem devolve parâmetro algum e simplesmente imprime todos os atributos do nosso funcionário. Dessa maneira, você não precisa ficar copiando e colando um monte de Console.WriteLine() para cada

mudança e teste que fizer com cada um de seus funcionários, você simplesmente vai fazer:

```
Funcionario f1 = new Funcionario();
//brincadeiras com f1....
f1.Mostra();
```

Veremos mais a frente o método ToString, que é uma solução muito mais elegante para mostrar a representação de um objeto como String, além de não jogar tudo pro Console.WriteLine (só se você desejar).

O esqueleto do método ficaria assim:

```
public class Funcionario
{
        // seus outros atributos e métodos
        public void Mostra()
        {
            Console.WriteLine("Nome: " + this.nome);
            // imprimir aqui os outros atributos...
        }
}
```

Construa dois funcionários com o new e compare-os com o ==. E se eles tiverem os mesmos atributos? Para isso você vai precisar criar outra referência:

```
Funcionario f1 = new Funcionario();
f1.nome = "Fiodor";
f1.salario = 100;
Funcionario f2 = new Funcionario();
f2.nome = "Fiodor";
f2.salario = 100;
if (f1 == f2)
{
      Console.WriteLine("iguais");
}
else
{
      Console.WriteLine("diferentes");
}
```

5) Crie duas referências para o **mesmo** funcionário, compare-os com o ==. Tire suas conclusões. Para criar duas referências pro mesmo funcionário:

```
Funcionario f1 = new Funcionario();
f1.nome = "Fiodor";
f1.salario = 100;
Funcionario f2 = f1;
```

O que acontece com o if do exercício anterior?

6) (opcional) Em vez de utilizar uma String para representar a data, crie uma outra classe, chamada Data. Ela possui 3 campos int, para dia, mês e ano. Faça com que

seu funcionário passe a usá-la. (é parecido com o último exemplo, em que a Conta passou a ter referência para um Cliente).

```
public class Funcionario
{
        Data dataDeEntrada; // qual é o valor default aqui?
        // seus outros atributos e métodos
}

public class Data
{
        int dia;
        int mes;
        int ano;
}
```

Modifique sua classe TestaFuncionario para que você crie uma Data e atribua ela ao Funcionario:

```
Funcionario f1 = new Funcionario();
//...
Data data = new Data(); // ligação!
f1.dataDeEntrada = data;
```

Faça o desenho do estado da memória quando criarmos um Funcionario.

7) (opcional) Modifique seu método mostra para que ele imprima o valor da dataDeEntrada daquele Funcionario:

```
public class Funcionario
{
        // seus outros atributos e métodos
        Data dataDeEntrada;
        public void mostra()
        {
            Console.WriteLine("Nome: " + this.nome);
            // imprimir aqui os outros atributos...
            Console.WriteLine("Dia: " + this.dataDeEntrada.dia);
            Console.WriteLine("Mês: " + this.dataDeEntrada.mes);
            Console.WriteLine("Ano: " + this.dataDeEntrada.ano);
        }
}
```

Teste-o.

Agora, o que acontece se chamarmos o método mostra antes de atribuirmos uma data para este Funcionario?

8) (opcional) O que acontece se você tentar acessar um atributo diretamente na classe? Como, por exemplo:

```
conta.saldo = 1234;
```

Esse código faz sentido? E este:

```
conta.Saca(50);
```

Faz sentido pedir para o esquema do conta sacar uma quantia?

2.13 – Desafios

1) Um método pode chamar ele mesmo. Chamamos isso de **recursão**. Você pode resolver a série de fibonacci usando um método que chama ele mesmo. O objetivo é você criar uma classe, que possa ser usada da seguinte maneira:

```
Fibonacci fibo = new Fibonacci();
int i = fibo.CalculaFibonacci(5);
Console.WriteLine(i);
```

Aqui imprimirá 8, já que este é o sexto número da série.

Este método CalculaFibonacci não pode ter nenhum laço, só pode chamar ele mesmo como método. Pense nele como uma função, que usa a própria função para calcular o resultado.

2) Por que o modo acima é extremamente mais lento para calcular a série do que o modo iterativo (que se usa um laço)?

3) Escreva o método recursivo novamente, usando apenas uma linha. Para isso, pesquise sobre o **operador condicional ternário**. (ternary operator)

2.14 - Fixando o conhecimento

O objetivo dos exercícios a seguir é fixar o conceito de classes e objetos, métodos e atributos. Dada a estrutura de uma classe, basta traduzí-la para a linguagem C# e fazer uso de um objeto da mesma em um programa simples.

Se você está com dificuldade em alguma parte desse capítulo, aproveite e treine tudo o que vimos até agora nos pequenos programas abaixo:

1) Programa 1 - Classe: Pessoa Atributos: nome, idade. Método: void FazAniversario(). Crie uma pessoa, coloque seu nome e idade iniciais, faça alguns aniversários (aumentando a idade) e imprima seu nome e sua idade.

2) Programa 2 - Classe: Porta Atributos: aberta, cor, dimensaoX, dimensaoY, dimensaoZ Métodos: void Abre(), void Fecha(), void Pinta(String s), boolean EstaAberta(). Crie uma porta, abra e feche a mesma, pinte-a de diversas cores, altere suas dimensões e use o método EstaAberta para verificar se ela está aberta.

3) Programa 3 - Classe: Casa Atributos: cor, porta1, porta2, porta3 Método: void Pinta(String s), int QuantasPortasEstaoAbertas(). Crie uma casa e pinte-a. Crie três portas e coloque-as na casa; abra e feche as mesmas como desejar.

Utilize o método QuantasPortasEstaoAbertas para imprimir o número de portas abertas.