UNIDADE 2 – ORIENTAÇÃO A OBJETOS

Nessa unidade veremos um pouco mais de Programação Orientada a Objetos.

Mas afinal o que é POO? Ela foi criada para tentar aproximar o mundo real e o mundo virtual: a ideia fundamental é tentar simular o mundo real dentro do computador. Para isso, nada mais natural do que utilizar objetos, afinal, nosso mundo é composto de objetos, certo?

Na Programação Orientada a Objetos, o programador (você) é responsável por moldar o mundo dos objetos, e definir como os objetos devem interagir entre si. Os objetos "conversam" uns com os outros através do envio de mensagens, e o papel principal do programador é definir quais serão as mensagens que cada objeto pode receber, e também qual a ação que o objeto deve realizar ao receber cada mensagem.

Toda essa complexidade e benefícios de Orientado a Objetos serão explicadas nos próximos capítulos.

SUMÁRIO

1 DOMÍNIO E APLICAÇÃO

Um domínio é composto pelas entidades, informações e processos relacionados a um determinado contexto. Uma aplicação pode ser desenvolvida para automatizar ou tornar factível as tarefas de um domínio. Portanto, uma aplicação é basicamente o "reflexo" de um domínio.

Para exemplificar, suponha que estamos interessados em desenvolver uma aplicação para facilitar as tarefas do cotidiano de um banco. Podemos identificar clientes, funcionários, agências e contas como entidades desse domínio. Assim como podemos identificar as informações e os processos relacionados a essas entidades.

Figura 1: Domínio bancário..

Mais Sobre

A identificação dos elementos de um domínio é uma tarefa difícil, pois depende fortemente do conhecimento das entidades, informações e processos que o compõem. Em geral, as pessoas que possuem esse conhecimento ou parte dele estão em contato constante com o domínio e não possuem conhecimentos técnicos para desenvolver uma aplicação.

Desenvolvedores de software buscam constantemente mecanismos para tornar mais eficiente o entendimento dos domínios para os quais eles devem desenvolver aplicações.

1.1 Objetos, Atributos e Métodos

As entidades identificadas no domínio devem ser representadas de alguma forma dentro da aplicação correspondente. Nas aplicações orientadas a objetos, as entidades são representadas por objetos.

- Uma aplicação orientada a objetos é composta por objetos.
- Em geral, um objeto representa uma entidade do domínio.

Para exemplificar, suponha que no domínio de um determinado banco exista um cliente chamado João. Dentro de uma aplicação orientada a objetos correspondente a esse domínio, deve existir um objeto para representar esse cliente.

Suponha que algumas informações do cliente João como nome, data de nascimento e sexo são importantes para o banco. Já que esses dados são relevantes para o domínio, o objeto que representa esse cliente deve possuir essas informações. Esses dados são armazenados nos atributos do objeto que representa o João.

- Um atributo é uma variável que pertence a um objeto.
- Os dados de um objeto são armazenados nos seus atributos.

O próprio objeto deve realizar operações de consulta ou alteração dos valores de seus atributos. Essas operações são definidas nos métodos do objeto. Os métodos também são utilizados para possibilitar interações entre os objetos de uma aplicação. Por exemplo, quando um cliente requisita um saque através de um caixa eletrônico do banco, o objeto que representa o caixa eletrônico deve interagir com o objeto que representa a conta do cliente.

- As tarefas que um objeto pode realizar são definidas pelos seus métodos.
- Um objeto é composto por atributos e métodos.

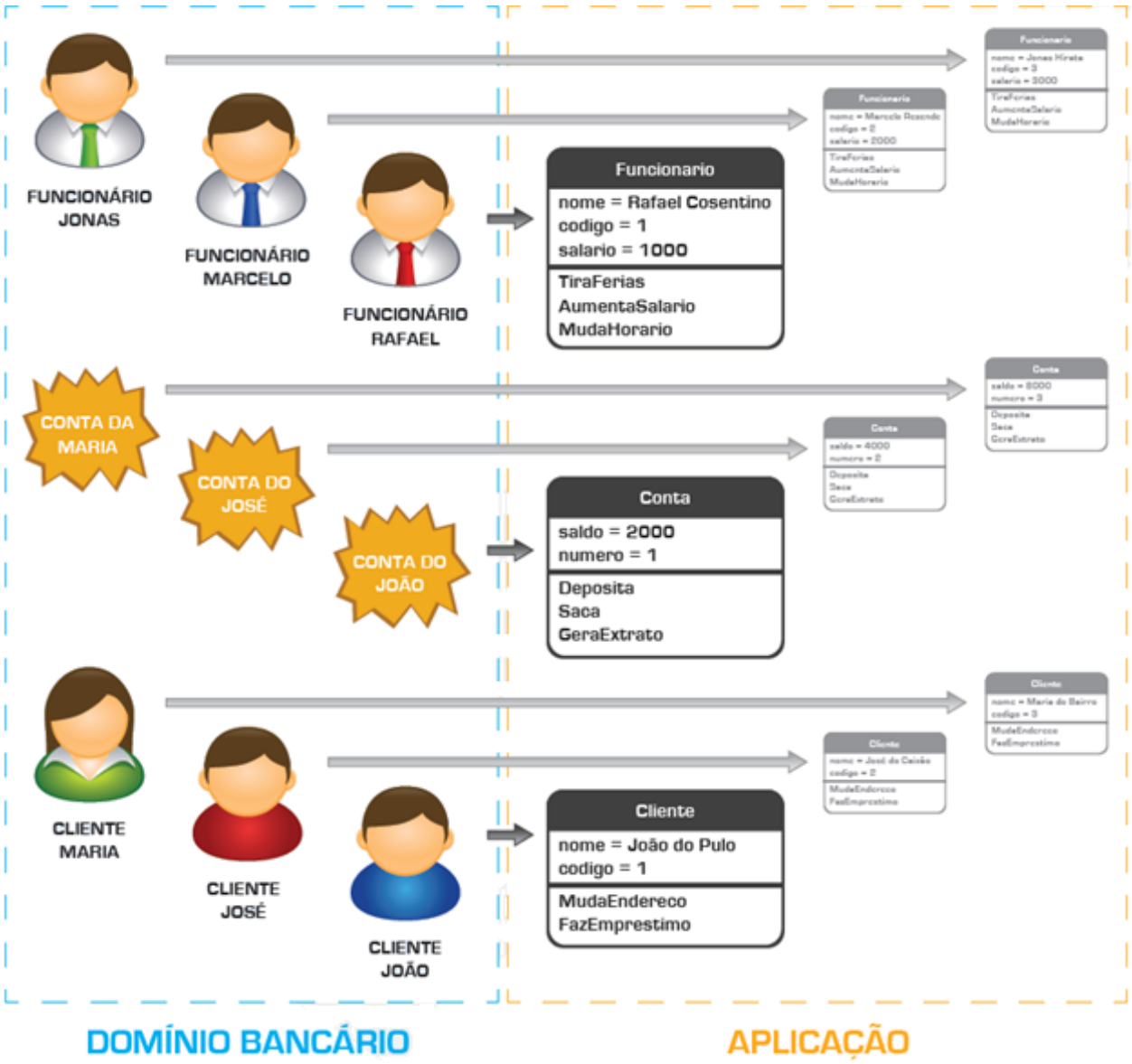

Figura 2: Mapeamento Domínio-Aplicação

Mais Sobre

Em geral, não é adequado utilizar o objeto que representa um determinado cliente para representar outro cliente do banco, pois os dados dos clientes podem ser diferentes. Dessa forma, para cada cliente do banco, deve existir um objeto dentro do sistema para representá-lo.

Mais Sobre

Os objetos não representam apenas coisas concretas como os clientes do banco. Eles também devem ser utilizados para representar coisas abstratas como uma conta de um cliente ou um serviço que o banco ofereça.

1.2 CLASSES

Antes de um objeto ser criado, devemos definir quais serão os seus atributos e métodos. Essa definição é realizada através de uma classe elaborada por um programador. A partir de uma classe, podemos construir objetos na memória do computador que executa a nossa aplicação.

Podemos representar uma classe através de diagramas UML. O diagrama UML de uma classe é composto pelo nome da mesma e pelos atributos e métodos que ela define. Todos os objetos criados a partir da classe Conta terão os atributos e métodos mostrados no diagrama UML. Os valores dos atributos de dois objetos criados a partir da classe Conta podem ser diferentes.

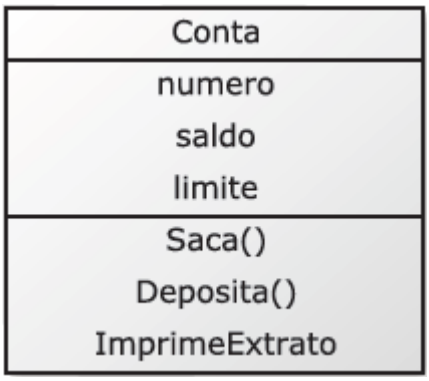

Figura 3: Diagrama UML da classe Conta.

Analogia

Um objeto é como se fosse uma casa ou um prédio. Para ser construído, precisa de um espaço físico. No caso dos objetos, esse espaço físico é algum trecho vago da memória do computador que executa a aplicação. No caso das casas e dos prédios, o espaço físico é algum terreno vazio.

Um prédio é construído a partir de uma planta criada por um engenheiro ou arquiteto. Para criar um objeto, é necessário algo semelhante a uma planta para que sejam “desenhados” os atributos e métodos que o objeto deve ter. Em orientação a objetos, a “planta” de um objeto é o que chamamos de classe.

Uma classe funciona como uma “receita” para criar objetos. Inclusive, vários objetos podem ser criados a partir de uma única classe. Assim como várias casas ou prédios poderiam ser construídos a partir de uma única planta; ou vários bolos poderiam ser preparados a partir de uma única receita; ou vários carros poderiam ser construídos a partir de um único projeto.

Figura 6: Diversas casas construídas a partir da mesma planta.

Figura 5: Diversos bolos preparados a partir da mesma receita.

Figura 4: Diversos carros construídos a partir do mesmo projeto.

Basicamente, as diferenças entre dois objetos criados a partir da classe Conta são os valores dos seus atributos. Assim como duas casas construídas a partir da mesma planta podem possuir características diferentes. Por exemplo, a cor das paredes.

Figura 7: Diversas casas com características diferentes.

1.3 CLASSES EM C#

O conceito de classe apresentado anteriormente é genérico e pode ser aplicado em diversas linguagens de programação. Mostraremos como a classe Conta poderia ser escrita utilizando a linguagem C#. Inicialmente, discutiremos apenas sobre os atributos. Os métodos serão abordados posteriormente.

```
1  public class Conta
2  {
3      public double saldo;
4      public double limite;
5      public int numero;
6  }
```

Tabela 1: Conta.cs

A classe C# Conta é declarada utilizando a palavra reservada class. No corpo dessa classe, são declaradas três variáveis que são os atributos que os objetos possuirão. Como a linguagem C# é estaticamente tipada, os tipos dos atributos são definidos no código. Os atributos saldo e limite são do tipo double, que permite armazenar números com casas decimais, e o atributo número é do tipo int, que permite armazenar números inteiros. O modificador public é adicionado em cada atributo para que eles possam ser acessados a partir de qualquer ponto do código. Discutiremos sobre esse e outros modificadores de visibilidade em capítulos posteriores.

Importante

Por convenção, os nomes das classes na linguagem C# devem seguir o padrão “pascal case” também conhecido como “upper camel case”.

1.3.1 Criando objetos em C#

Após definir a classe Conta, podemos criar objetos a partir dela. Esses objetos devem ser alocados na memória RAM do computador. Felizmente, todo o processo de alocação do objeto na memória é gerenciado pela máquina virtual. O gerenciamento da memória é um dos recursos mais importantes oferecidos pela máquina virtual.

Do ponto de vista da aplicação, basta utilizar um comando especial para criar objetos e a máquina virtual se encarrega do resto. O comando para criar objetos é o **new**.

```
1  public class TestaConta
2  {
3      0 references
4      private static void Main()
5      {
6          new Conta();
7      }
   }
```

Tabela 2: TestaConta.cs

A linha com o comando new poderia ser repetida cada vez que desejássemos criar (instanciar) um objeto da classe Conta. A classe TestaConta serve apenas para colocarmos o método Main, que é o ponto de partida da aplicação.

```
1  public class TestaConta
2  {
3      0 references
4      private static void Main()
5      {
6          //criando novos objetos
8          new Conta();
9          new Conta();
10         new Conta();
11     }
   }
```

Tabela 3: TestaConta.cs

Analogia

Chamar o comando new passando uma classe C# é como se estivéssemos contratando uma construtora passando a planta da casa que queremos construir. A construtora se encarrega de construir a casa para nós de acordo com a planta. Assim como a máquina virtual se encarrega de construir o objeto na memória do computador.

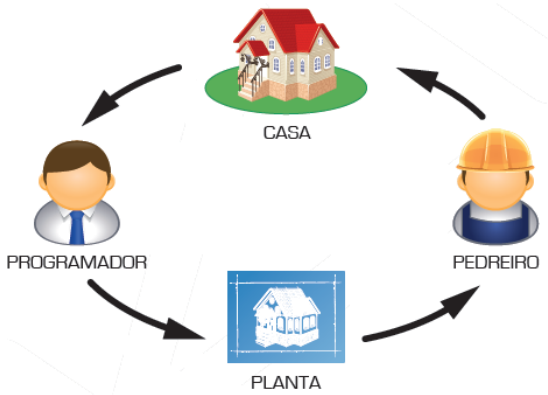

Figura 8: Construindo casas

1.4 Referências

Todo objeto possui uma referência. A referência de um objeto é a única maneira de acessar os seus atributos e métodos. Dessa forma, devemos guardar as referências dos objetos que desejamos utilizar.

Analogia

A princípio, podemos comparar a referência de um objeto com o endereço de memória desse objeto. De fato, essa comparação simplifica o aprendizado. Contudo, o conceito de referência é mais amplo. Uma referência é o elemento que permite que um determinado objeto seja acessado.

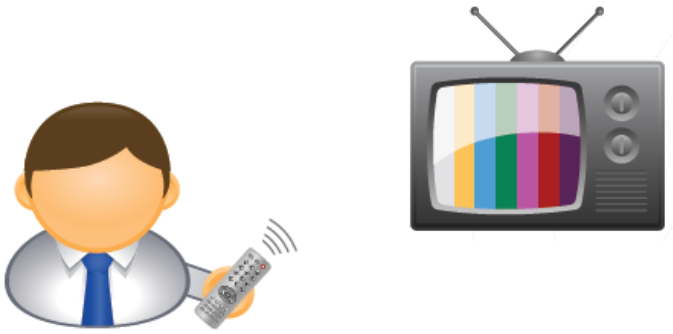

Figura 9: Controle Remoto

1.4.1 Referências em C#

Ao utilizar o comando new, um objeto é alocado em algum lugar da memória. Para que possamos acessar esse objeto, precisamos de sua referência. O comando new devolve a referência do objeto que foi criado.

Para guardar as referências devolvidas pelo comando new, devemos utilizar variáveis não primitivas.

```
1   Conta conta =  new Conta();
```

Tabela 4: Criando um objeto e guardando a referência.

No código C# acima, a variável **referência** receberá a referência do objeto criado pelo comando new. Essa variável é do tipo Conta. Isso significa que ela só pode armazenar referências de objetos do tipo Conta.

1.5 MANIPULANDO ATRIBUTOS

Podemos alterar ou acessar os valores guardados nos atributos de um objeto se tivermos a referência a esse objeto. Os atributos são acessados pelo nome. No caso específico da linguagem C#, a sintaxe para acessar um atributo utiliza o operador ".".

No código acima, o atributo saldo recebe o valor 1000.0. O atributo limite recebe o valor 500 e o número recebe o valor 1. Depois, os valores são impressos na tela através do comando **System.Console.WriteLine**.

```
1   Conta referecia = new Conta();
2
3   referecia.saldo = 1000.0;
4   referecia.limite = 500.0;
5   referecia.numero = 1;
6
7   System.Console.WriteLine(referecia.saldo);
8   System.Console.WriteLine(referecia.limite);
    System.Console.WriteLine(referecia.numero);
```

Tabela 5: Alterando e acessando os atributos de um objeto.

1.6 VALORES PADRÃO

Poderíamos instanciar um objeto e utilizar seus atributos sem inicializá-los explicitamente, pois os atributos são inicializados com valores padrão. Os atributos de tipos numéricos são inicializados com 0, os atributos do tipo boolean são inicializados com false e os demais atributos com null (referência vazia).

```
1  public class TestaConta
2  {
3      0 references
4      private static void Main()
5      {
6          Conta referecia = new Conta();
7
8          // imprime 0
9          System.Console.WriteLine(referecia.limite);
10     }
11 }
```

Tabela 6: TestaConta.cs.

```
1  public class Conta
2  {
3      public double limite;
4  }
```

Tabela 7: Conta.cs.

A inicialização dos atributos com os valores padrão ocorre na instanciação, ou seja, quando o comando new é utilizado. Dessa forma, todo objeto "nasce" com os valores padrão. Em alguns casos, é necessário trocar esses valores. Para trocar o valor padrão de um atributo, devemos inicializá-lo na declaração. Por exemplo, suponha que o limite padrão das contas de um banco seja R$ 500. Nesse caso, seria interessante definir esse valor como padrão para o atributo limite.

```
1  public class Conta
2  {
3      public double limite = 500;
4  }
```

Tabela 8: Conta.cs.

```
1  public class TestaConta
2  {
3      0 references
4      private static void Main()
5      {
6          Conta referecia = new Conta();
7
8          // imprime 500
9          System.Console.WriteLine(referecia.limite);
10     }
11 }
```

Tabela 9: TestaConta.cs.

1.7 EXERCÍCIOS DE FIXAÇÃO

1) Na solução AcademiaProgramdaor crie outro projeto com o nome de Unidade2. Dentro desse projeto, adicione uma pasta chamada **Orientacao_a_Objetos** para os arquivos desenvolvidos nesta Unidade.

2) Implemente uma classe para definir os objetos que representarão os clientes de um banco. Essa classe deve declarar dois atributos: um para os nomes e outro para os códigos dos clientes. Adicione o seguinte arquivo na pasta **Orientacao_a_Objetos**:

```
1  class Cliente
2  {
3      public string nome;
4      public int codigo;
5  }
```

Tabela 10: Cliente.cs.

3) Faça um teste criando dois objetos da classe Cliente. Adicione o seguinte arquivo na pasta Orientacao_a_Objetos:

```
1  class TestaCliente
2  {
3      0 references
4      private static void Main()
5      {
6          Cliente c1 = new Cliente();
7          c1.nome = " Thiago Sartor ";
8          c1.codigo = 1;
9
10         Cliente c2 = new Cliente();
11         c2.nome = " Alexandre Rech ";
12         c2.codigo = 2;
13
14         System.Console.WriteLine(c1.nome);
15         System.Console.WriteLine(c1.codigo);
16
17         System.Console.WriteLine(c2.nome);
18         System.Console.WriteLine(c2.codigo);
19     }
20 }
```

Tabela 11: TestaCliente.cs.

Compile e execute a classe TestaCliente.

4) Os bancos oferecem aos clientes a possibilidade de obter um cartão de crédito que pode ser utilizado para fazer compras. Um cartão de crédito possui um número e uma data de validade. Crie uma classe para modelar os objetos que representarão os cartões de crédito. Adicione o seguinte arquivo na pasta Orientacao_a_Objetos.

```
1  class CartaoDeCredito
2  {
3      public int numero;
4      public string dataDeValidade;
5  }
```

Tabela 12: CartaoCredito.cs.

5) Faça um teste criando dois objetos da classe CartaoDeCredito. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta Orientacao_a_Objetos:

```
1  class TestaCartaoDeCredito
2  {
3      0 references
4      private static void Main()
5      {
6          CartaoDeCredito cdc1 = new CartaoDeCredito();
7          cdc1.numero = 114532543;
8          cdc1.dataDeValidade = "12/17";
9
10         CartaoDeCredito cdc2 = new CartaoDeCredito();
11         cdc2.numero = 098765676;
12         cdc2.dataDeValidade = "12/18";
13
14
15         System.Console.WriteLine(cdc1.numero);
16         System.Console.WriteLine(cdc1.dataDeValidade);
17
18         System.Console.WriteLine(cdc2.numero);
           System.Console.WriteLine(cdc2.dataDeValidade);
       }
   }
```

Tabela 13: TestaCartaoDeCredito.cs.

Compile e execute a classe TestaCartaoDeCredito.

Observação: Só existe um método principal(Main) em um projeto. Então o Main do TesteCliente deve estar dessa forma, "Main_", para que o Main do TesteCartaoDeCredito posso compilar.

6) As agências do banco possuem número. Crie uma classe para definir os objetos que representarão as agências.

```
1  class Agencia
2  {
3      public int numero;
4  }
```

Tabela 14: Agencia.cs.

7) Faça um teste criando dois objetos da classe Agencia. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta Orientacao_a_Objetos:

```
1  class TestaAgencia
2  {
3      0 references
4      private static void Main()
5      {
6          Agencia a1 = new Agencia();
7          a1.numero = 1234;
8
9          Agencia a2 = new Agencia();
10         a2.numero = 5678;
11
12         Console.WriteLine(a1.numero);
13
14         Console.WriteLine(a2.numero);
15     }
16 }
```

Tabela 15: TesteAgencia.cs.

8) As contas do banco possuem número, saldo e limite. Crie uma classe para definir os objetos que representarão as contas.

```
1  class Conta
2  {
3      public int numero;
4      public double saldo;
5      public double limite;
6  }
```

Tabela 16: Conta.cs.

9) Faça um teste criando dois objetos da classe Conta. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta Orientacao_a_Objetos:

```
1  class TestaConta
2  {
3      0 references
4      private static void Main()
5      {
6          Conta c1 = new Conta();
7          c1.numero = 1234;
8          c1.saldo = 1000;
9          c1.limite = 500;
10
11         Conta c2 = new Conta();
12         c2.numero = 5678;
13         c2.saldo = 2000;
14         c2.limite = 250;
15
16         Console.WriteLine(c1.numero);
17         Console.WriteLine(c1.saldo);
18         Console.WriteLine(c1.limite);
19
20         Console.WriteLine(c2.numero);
21         Console.WriteLine(c2.saldo);
22         Console.WriteLine(c2.limite);
23     }
24 }
```

Tabela 17: TesteConta.cs.

10) Faça um teste que imprima os atributos de um objeto da classe Conta logo após a sua criação.

```
1  class TestaValoresPadrao
2  {
3      0 references
4      private static void Main()
5      {
6          Conta c = new Conta();
7
8          Console.WriteLine(c.numero);
9          Console.WriteLine(c.saldo);
10         Console.WriteLine(c.limite);
11     }
12 }
```

Tabela 18: TestaValoresPadrao.cs.

11) Altere a classe Conta para que todos os objetos criados a partir dessa classe possuam R$ 100 de limite inicial.

```
1  class Conta
2  {
3      public int numero;
4      public double saldo;
5
6      //Adiciona a linha abaixo
7      public double limite = 100;
   }
```

Tabela 19: Conta.cs

.

1.8 Exercícios Complementares

1) Implemente uma classe chamada Aluno na pasta **Orientacao_a_Objetos** para definir os objetos que representarão os alunos de uma escola. Essa classe deve declarar três atributos: o primeiro para o nome, o segundo para o RG e o terceiro para a data de nascimento dos alunos.

2) Faça uma classe chamada TestaAluno e crie dois objetos da classe Aluno atribuindo valores a eles. A classe também deve mostrar na tela as informações desses objetos.

3) Em uma escola, além dos alunos temos os funcionários, que também precisam ser representados em nossa aplicação. Então implemente outra classe na pasta **Orientacao_a_Objetos** chamada Funcionário que contenha três atributos: o primeiro para o nome e o segundo para o cargo o terceiro cargo dos funcionários.

4) Faça uma classe chamada TestaFuncionario e crie dois objetos da classe Funcionario atribuindo valores a eles. Mostre na tela as informações desses objetos.

5) Em uma escola, os alunos precisam ser divididos por turmas, que devem ser representadas dentro da aplicação. Implemente na pasta **Orientacao_a_Objetos** uma classe chamada Turma que contenha quatro atributos: o primeiro para o período, o segundo para definir a série, o terceiro para sigla e o quarto para o tipo de ensino.

6) Faça uma classe chamada TestaTurma para criar dois objetos da classe Turma. Adicione informações a eles e depois mostre essas informações na tela.

1.9 Relacionamentos: Associação, Agregação e Composição

Todo cliente do banco pode adquirir um cartão de crédito. Suponha que um cliente adquira um cartão de crédito. Dentro do sistema do banco, deve existir um objeto que represente o cliente e outro que represente o cartão de crédito. Para expressar a relação entre o cliente e o cartão de crédito, algum vínculo entre esses dois objetos deve ser estabelecido.

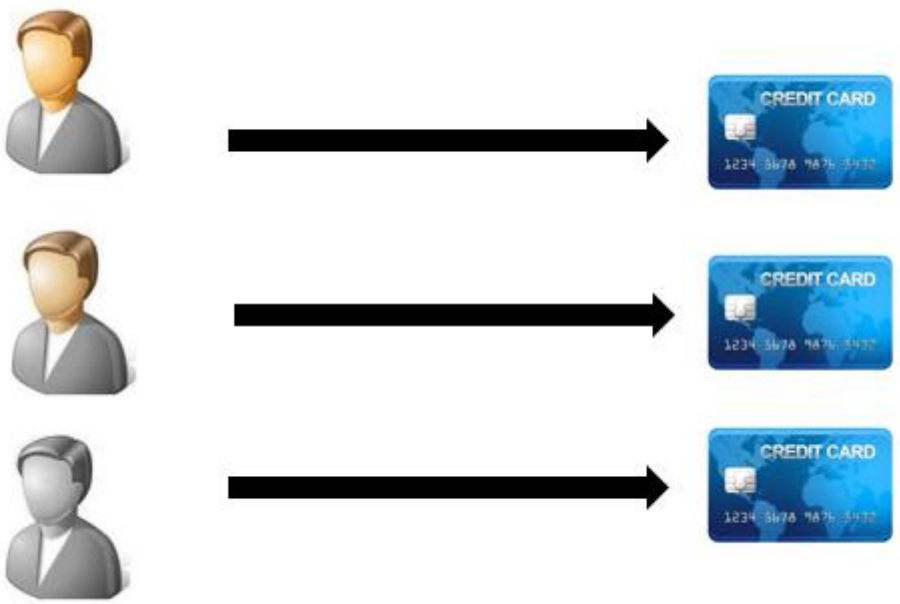

Figura 8: Clientes e cartões.

Duas classes deveriam ser criadas: uma para definir os atributos e métodos dos clientes e outra para os atributos e métodos dos cartões de crédito. Para expressar o relacionamento entre cliente e cartão de crédito, podemos adicionar um atributo do tipo Cliente na classe CartaoDeCredito.

```
1  class Cliente
2  {
3      public string nome;
4  }
```

Tabela 20: Cliente.cs

```
1  class CartaoDeCredito
2  {
3      public int numero;
4      public string dataDeValidade;
5      public Cliente cliente;
   }
```

Tabela 21: CartaoDeCredito.cs

Esse tipo de relacionamento é chamado de **Agregação**. Há uma notação gráfica na linguagem UML para representar uma agregação. Veja o diagrama abaixo.

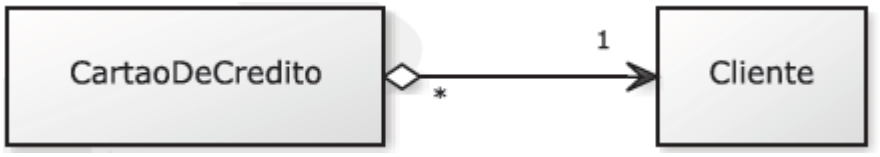

Figura 9: Agregação entre clientes e cartões de crédito

No relacionamento entre cartão de crédito e cliente, um cartão de crédito só pode se relacionar com um único cliente. Por isso, no diagrama acima, o número 1 é colocado ao lado da classe Cliente. Por outro lado, um cliente pode se relacionar com muitos cartões de crédito. Por isso, no diagrama acima, o caractere "*" é colocado ao lado da classe CartaoDeCredito.

O relacionamento entre um objeto da classe Cliente e um objeto da classe CartaoDeCredito só é concretizado quando a referência do objeto da classe Cliente é armazenada no atributo cliente do objeto da classe CartaoDeCredito. Depois de relacionados, podemos acessar, indiretamente, os atributos do cliente através da referência do objeto da classe CartaoDeCredito.

```
1  // Criando um objeto de cada classe
2  CartaoDeCredito cdc = new CartaoDeCredito();
3  Cliente c = new Cliente();
4
5  // Ligando os objetos
6  cdc.cliente = c;
7
8  // Acessando o nome do cliente
   cdc.cliente.nome = "Thiago Sartor";
```

Tabela 22: Concretizando uma agregacão

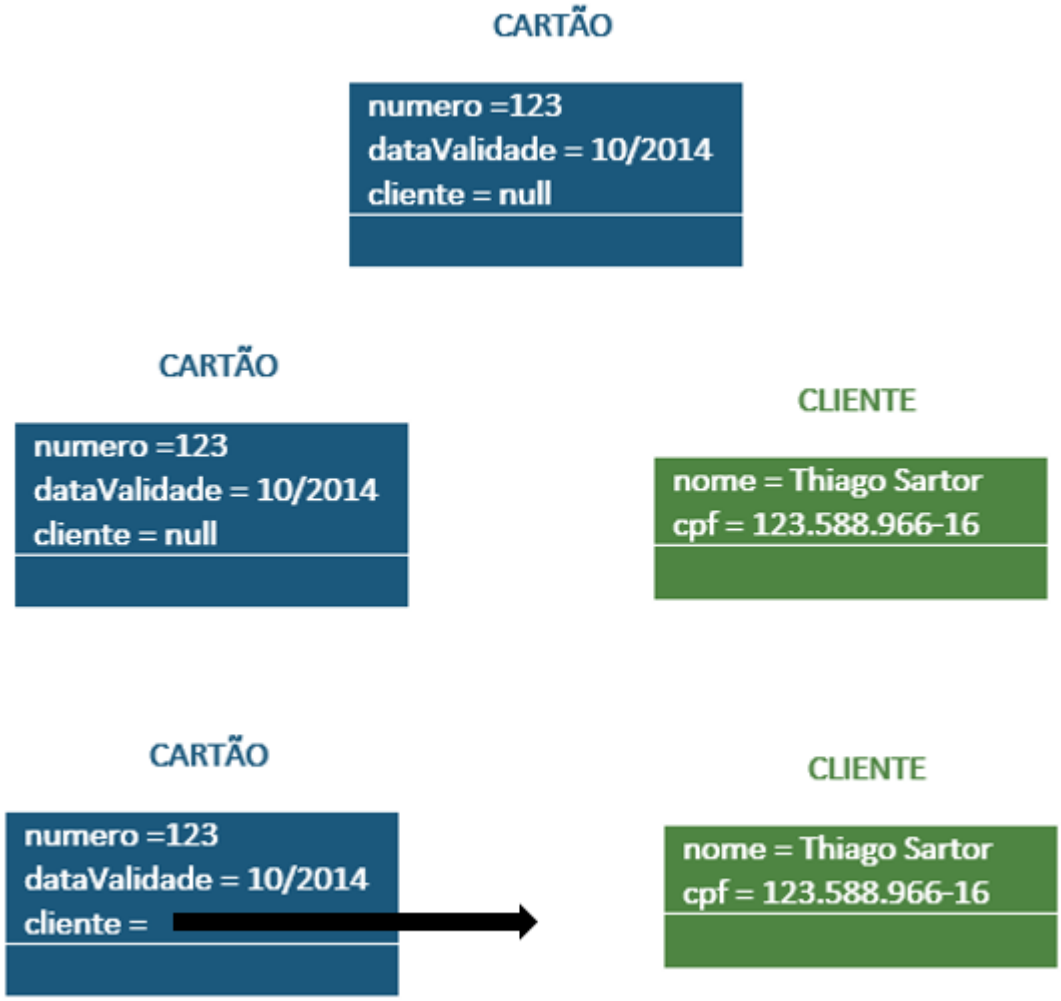

Figura 10: Conectando um cliente e um cartão

1.10 Exercícios de Fixação

12) Defina um vínculo entre os objetos que representam os clientes e os objetos que representam os cartões de crédito. Para isso, você deve alterar a classe CartaoDeCredito.

```
1  class CartaoDeCredito
2  {
3      public int numero;
4      public string dataDeValidade;
5
6      public Cliente cliente;
   }
```

Tabela 24: CartaoDeCredito.cs

13) Teste o relacionamento entre clientes e cartões de crédito.

```
1  class TestaCartaoDeCredito
2  {
3      0 references
4      private static void Main()
5      {
6          //Criando objetos
7          CartaoDeCredito cdc1 = new CartaoDeCredito();
8          cdc1.numero = 114532543;
9          cdc1.dataDeValidade = "12/17";
10
11         // Populando objetos
12         Cliente c = new Cliente();
13         c.codigo = 234324;
14         c.nome = "Thiago Saretor";
15
16         // Populando objetos
17         CartaoDeCredito cdc2 = new CartaoDeCredito();
18         cdc2.numero = 098765676;
19         cdc2.dataDeValidade = "12/18";
20
21         // Ligando os objetos
22         cdc1.cliente = c;
23
24
25         System.Console.WriteLine(cdc2.numero);
26         System.Console.WriteLine(cdc2.dataDeValidade);
27         System.Console.WriteLine(cdc2.cliente.nome);
28
29     }
30 }
```

Tabela 25:TestaClienteECartao.cs

Compile e execute a classe TestaClienteECartao.

14) Defina um vínculo entre os objetos que representam as agências e os objetos que representam as contas. Para isso, você deve alterar a classe Conta.

```
1  class ContaEAgencia
2  {
3      public int numero;
4      public double saldo;
5      public double limite = 100;
6
7      //Adiciona a linha abaixo
8      public Agencia agencia;
9  }
```

Tabela 26:ContaEAgencia.cs

15) Teste o relacionamento entre contas e agências.

```
1  class TestaContaEAgencia
2  {
3      0 references
4      private static void Main()
5      {
6          //Criando objetos
7          Agencia a = new Agencia();
8          a.numero = 2314324;
9
10         // Carregando dados
11         ContaEAgencia c1 = new ContaEAgencia();
12         c1.numero = 1234;
13         c1.saldo = 1000;
14         c1.limite = 500;
15
16         //Ligando os objetos
17         c1.agencia = a;
18
19         Console.WriteLine(c1.numero);
20         Console.WriteLine(c1.saldo);
21         Console.WriteLine(c1.limite);
22         Console.WriteLine(c1.agencia.numero);
23     }
24 }
```

Tabela 27: TestaContaEAgencia.cs

Compile e execute a classe TestaContaEAgencia.

1.11 Exercícios Complementares

7) Defina um vínculo entre os alunos e as turmas, criando na classe Aluno um atributo do tipo Turma.

8) Teste o relacionamento entre os alunos e as turmas, criando um objeto de cada classe e atribuindo valores a eles. Exiba na tela os valores que estão nos atributos da turma através do objeto da classe Aluno.

1.12 MÉTODOS

No banco, é possível realizar diversas operações em uma conta: depósito, saque, transferência, consultas e etc. Essas operações podem modificar ou apenas acessar os valores dos atributos dos objetos que representam as contas.

Essas operações são realizadas em **métodos** definidos na própria classe Conta. Por exemplo, para realizar a operação de depósito, podemos acrescentar o seguinte método na classe Conta.

```
1  void Deposita(double valor)
2  {
3      //implementação
4  }
```

Tabela 28: Estrutura de um método

Podemos dividir um método em quatro partes:

Nome: É utilizado para chamar o método. Na linguagem C#, é uma boa prática definir os nomes dos métodos utilizando a convenção "Camel Case" com a primeira letra maiúscula.

Lista de Parâmetros: Define os valores que o método deve receber. Métodos que não devem receber nenhum valor possuem a lista de parâmetros vazia.

Corpo: Define o que acontecerá quando o método for chamado.

Retorno: A resposta que será devolvida ao final do processamento do método. Quando um método não devolve nenhuma resposta, ele deve ser marcado coma palavra reservada **void**.

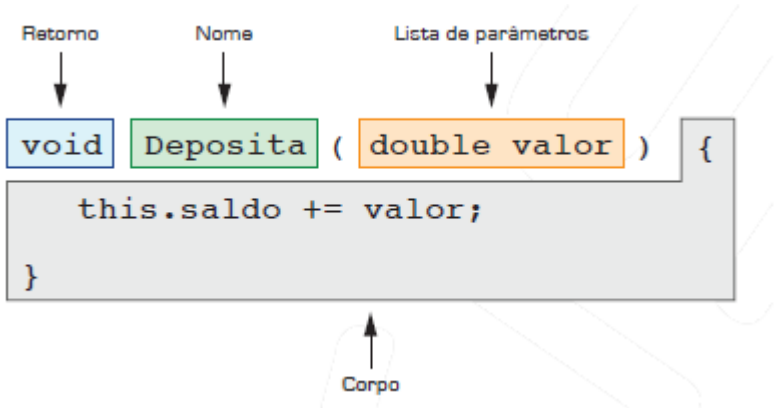

Figura 12: Estrutura de um método

Para realizar um depósito, devemos chamar o método `Deposita()` através da referência do objeto que representa a conta que terá o dinheiro creditado.

```
1  // Referência de um objeto
2  Conta c = new Conta();
3
4  // Chamando o método Deposita()
5  c.Deposita(1000);
```

Tabela 29: Chamando o método Deposita()

Normalmente, os métodos acessam ou alteram os valores armazenados nos atributos dos objetos. Por exemplo, na execução do método `Deposita()`, é necessário alterar o valor do atributo saldo do objeto que foi escolhido para realizar a operação.

Dentro de um método, para acessar os atributos do objeto que está processando o método, devemos utilizar a palavra reservada **this**.

```
1  public void Deposita(double valor)
2  {
3      this.saldo += valor;
4  }
```

Tabela 30: Utilizando o this para acessar e/ou modificar um atributo

O método `Deposita()`, não possui nenhum retorno lógico. Por isso, foi marcado com void. Mas, para outros métodos, pode ser necessário definir um tipo de retorno específico. Considere, por exemplo, um método para realizar a operação que consulta o saldo disponível das contas. Suponha também que o saldo disponível é igual a soma do saldo e do limite. Então, esse método deve somar os atributos saldo e limite e devolver o resultado. Por outro lado, esse método não deve receber nenhum valor, pois todas as informações necessárias para realizar a operação estão nos atributos dos objetos que representam as contas.

```
1  public double ConsultaSaldoDisponivel()
2  {
3      return this.saldo + this.limite;
4  }
```

Tabela 31: Método com retorno double

Ao chamar o método `ConsultaSaldoDisponivel()` a resposta pode ser armazenada em uma variável do tipo `double`.

```
1  // Referência de um objeto
2  Conta c = new Conta();
3
4  c.Deposita(1000);
5  //Armazenando a resposta de um método em uma variável
6  double saldoDisponivel = c.ConsultaSaldoDisponivel();
7
   Console.WriteLine("Saldo Disponível : " + saldoDisponivel);
```

Tabela 32: Armazenando a resposta de um método

1.13 Exercícios de Fixação

16) Acrescente alguns métodos na classe Conta para realizar as operações de deposito, saque, impressão de extrato e consulta do saldo disponível.

```
1  class Conta
2  {
3      public int numero;
4      public double saldo;
5      public double limite = 100;
6      public Agencia agencia;
7
8      // Adicione os métodos abaixo
9
10     1 reference
11     public void Deposita(double valor)
12     {
13         this.saldo += valor;
14     }
15
16     1 reference
17     public void Saca(double valor)
18     {
19         this.saldo -= valor;
20     }
21
22     2 references
23     public void ImprimeExtrato()
24     {
25         Console.WriteLine(" SALDO : " + this.saldo);
26     }
27
28     1 reference
29     public double ConsultaSaldoDisponivel()
30     {
31         return this.saldo + this.limite;
32     }
33 }
```

Tabela 33: Conta.cs

17) Teste os métodos da classe Conta.

```
1  class TestaConta
2  {
3      0 references
4      private static void Main()
5      {
6          Conta c = new Conta();
7
8          c.Deposita(1000);
9          c.ImprimeExtrato();
10
11         c.Saca(100);
12         c.ImprimeExtrato();
13
14         double saldoDisponivel = c.ConsultaSaldoDisponivel();
15         Console.WriteLine(" SALDO DISPONÍVEL : " + saldoDisponivel);
16     }
17 }
```

Tabela 34: TestaMetodosConta.cs

1.14 Exercícios Complementares

9) Sabendo que qualquer empresa possui funcionários, crie uma classe chamada Funcionário para representá-los. Acrescente os atributos nome e salario a essa classe. Além disso, você deve criar dois métodos: um para aumentar o salário e outro para consultar os dados dos funcionários.

10) Crie uma classe chamada TestaFuncionario para testar os métodos de um objeto da classe Funcionario.

1.15 SOBRECARGA (OVERLOADING)

Os clientes dos bancos costumam consultar periodicamente informações relativas às suas contas. Geralmente, essas informações são obtidas através de extratos. No sistema do banco, os extratos podem ser gerados por métodos da classe Conta.

```
1  class Conta
2  {
3      public double saldo;
4      public double limite;
5
6      0 references
7      public void ImprimeExtrato(int dias)
8      {
9          // extrato
10     }
```

Tabela 35: Conta.cs

O método `ImprimeExtrato`() recebe a quantidade de dias que deve ser considerada para gerar o extrato da conta. Por exemplo, se esse método receber o valor 30 então ele deve gerar um extrato com as movimentações dos últimos 30 dias.

Em geral, extratos dos últimos 15 dias atendem as necessidades dos clientes. Dessa forma, poderíamos acrescentar um método na classe Conta para gerar extratos com essa quantidade fixa de dias.

```
1  class Conta
2  {
3      public double saldo;
4      public double limite;
5
6      2 references
7      public void ImprimeExtrato()
8      {
9          // extrato dos últimos 15 dias
10     }
11
12     0 references
13     public void ImprimeExtrato(int dias)
14     {
15         // extrato
16     }
17 }
```

Tabela 36: Conta.cs

O primeiro método não recebe parâmetros pois ele utilizará uma quantidade de dias padrão definida pelo banco para gerar os extratos (15 dias).

O segundo recebe um valor inteiro como parâmetro e deve considerar essa quantidade de dias para gerar os extratos.

Os dois métodos possuem o mesmo nome e lista de parâmetros diferentes. Quando dois ou mais métodos são definidos na mesma classe com o mesmo nome, dizemos que houve uma sobrecarga de métodos. Uma sobrecarga de métodos só é válida se as listas de parâmetros dos métodos são diferentes entre si.

No caso dos dois métodos que geram extratos, poderíamos evitar repetição de código fazendo um método chamar o outro.

```
1  class Conta
2  {
3      public double saldo;
4      public double limite;
5
6
7      1 reference
8      public void ImprimeExtrato(int dias)
9      {
10         // extrato
11     }
12
13     2 references
14     public void ImprimeExtrato()
15     {
16         this.ImprimeExtrato(15);
17     }
18 }
```

Tabela 37: Conta.cs

1.16 Exercícios de Fixação

18) Crie uma classe chamada Gerente para definir os objetos que representarão os gerentes do banco. Defina dois métodos de aumento salarial nessa classe. O primeiro deve aumentar o salário com uma taxa fixa de 10%. O segundo deve aumentar o salário com uma taxa variável.

```
1  class Gerente
2  {
3      public string nome;
4      public double salario;
5
6      0 references
7      public void AumentoSalario()
8      {
9          this.AumentoSalario(0.1);
10     }
11
12     1 reference
13     public void AumentoSalario(double taxa)
14     {
15         this.salario += this.salario*taxa;
16     }
17 }
```

Tabela 38: Gerente.cs

19) Teste os métodos de aumento salarial definidos na classe Gerente.

```
1  class TestaGerente
2  {
3      public string nome;
4      public double salario;
5
6      0 references
7      public void AumentoSalario()
8      {
9          this.AumentoSalario(0.1);
10     }
11
12     1 reference
13     public void AumentoSalario(double taxa)
14     {
15         this.salario += this.salario*taxa;
16     }
17 }
```

Tabela 39: TesteGerente.cs

Compile e execute a classe `TestaGerente`.

1.17 CONSTRUTORES

No domínio de um banco, todo cartão de crédito deve possuir um número. Toda agência deve possuir um número. Toda conta deve estar associada a uma agência.

Após criar um objeto para representar um cartão de crédito, poderíamos definir um valor para o atributo número. De maneira semelhante, podemos definir um número para um objeto da classe Agencia e uma agência para um objeto da classe Conta.

```
1  CartaoDeCredito cdc = new CartaoDeCredito();
2  cdc.numero = 1232133213;
```

Tabela 40: Definindo um número para um cartão de crédito

```
1  Agencia a = new Agencia();
2  a.numero = 3242342;
```

Tabela 41: Definindo um número para uma agência

```
1  Conta c = new Conta();
2  c.agencia = a;
```

Tabela 42: Definindo uma agência para uma conta

Definir os valores dos atributos obrigatórios de um objeto logo após a criação dele resolveria as restrições do sistema do banco. Porém, nada garante que todos os desenvolvedores sempre lembrem de inicializar esses valores.

Para não correr esse risco, podemos utilizar construtores. Um construtor permite que um determinado trecho de código seja executado toda vez que um objeto é criado, ou seja, toda vez que o operador new é chamado. Assim como os métodos, os construtores podem receber parâmetros. Contudo, diferentemente dos métodos, os construtores não devolvem resposta.

Em C#, um construtor deve ter o mesmo nome da classe na qual ele foi definido.

```
1  class CartaoDeCredito
2  {
3      public int numero;
4
5      4 references
6      public CartaoDeCredito(int numero)
7      {
8          this.numero = numero;
9      }
   }
```

Tabela 43: CartaoDeCredito.cs

```
1  class Agencia
2  {
3      public int numero;
4
5      3 references
6      public Agencia(int numero)
7      {
8          this.numero = numero;
9      }
10 }
```

Tabela 44: Agencia.cs

```
1  class Conta
2  {
3      public Agencia agencia;
4
5      2 references
6      public Conta(Agencia agencia)
7      {
8          this.agencia = agencia;
9      }
10 }
```

Tabela 45: Conta.cs

Na criação de um objeto com o comando new, os argumentos passados devem ser compatíveis com a lista de parâmetros de algum construtor definido na classe que está sendo instanciada. Caso contrário, um erro de compilação ocorrerá para avisar o desenvolvedor dos valores obrigatórios que devem ser passados para criar um objeto.

```
1  // Passando corretamente os parâmetros para os construtores
2  CartaoDeCredito cdc = new CartaoDeCredito(11111);
3
4  Agencia a = new Agencia(12323);
5
6  Conta c = new Conta(a);
```

Tabela 46: Construtores

```
1  //ERRO DE COMPILAÇÃO
2  CartaoDeCredito cdc = new CartaoDeCredito();
3  cdc.numero = 1232133213;
4
5  //ERRO DE COMPILAÇÃO
6  Agencia a = new Agencia();
7  a.numero = 3242342;
8
9  //ERRO DE COMPILAÇÃO
10 Conta c = new Conta();
11 c.agencia = a;
```

Tabela 47: Construtores

1.17.1 Construtor Padrão

Toda vez que um objeto é criado, um construtor da classe correspondente deve ser chamado. Mesmo quando nenhum construtor for definido explicitamente, há um construtor padrão que será inserido pelo próprio compilador. O construtor padrão não recebe parâmetros e será inserido sempre que o desenvolvedor não definir pelo menos um construtor explicitamente.

Portanto, para instanciar uma classe que não possui construtores definidos no código fonte, devemos utilizar o construtor padrão, já que este é inserido automaticamente pelo compilador.

```
1  class Conta
2  {
3
4  }
```

Tabela 48: Conta.cs

```
1  // Chamando o construtor padrão
2  Conta c = new Conta();
```

Tabela 49: Utilizando o construtor padrão

Lembrando que o construtor padrão só será inserido pelo compilador se nenhum construtor for definido no código fonte. Dessa forma, se você adicionar um construtor com parâmetros então não poderá utilizar o comando new sem passar argumentos, pois um erro de compilação ocorrerá.

```
1  class Agencia
2  {
3      public int numero;
4
5      3 references
6      public Agencia(int numero)
7      {
8          this.numero = numero;
9      }
10 }
```

Tabela 50: Agencia.cs

```
1  Agencia a = new Agencia();
2  a.numero = 1234;
```

Tabela 51: Chamando um construtor sem araumentos

1.17.2 Sobrecarga de Construtores

O conceito de sobrecarga de métodos pode ser aplicado para construtores. Dessa forma, podemos definir diversos construtores na mesma classe.

```
1  class Pessoa
2  {
3      public string rg;
4      public int cpf;
5
6      0 references
7      public Pessoa(string rg)
8      {
9          this.rg = rg;
10     }
11
12     0 references
13     public Pessoa(int cpf)
14     {
15         this.cpf = cpf;
16     }
17 }
```

Tabela 52: Pessoa.cs

Quando dois construtores são definidos, há duas opções no momento de utilizar o comando **new**.

```
1  //Chamando o primeiro construtor
2  Pessoa p1 =new Pessoa("23423423");
3
4  //Chamando o segundo construtor
5  Pessoa p2 = new Pessoa(23423423);
```

Tabela 53: Utilizando dois construtores diferentes

1.17.3 Construtores chamando Construtores

Assim como podemos encadear, métodos também podemos encadear construtores.

```
1  class Conta
2  {
3      public int numero;
4      public double limite;
5
6      2 references
7      public Conta(int numero)
8      {
9          this.numero = numero;
10     }
11
12     1 reference
13     public Conta(int numero, double limite) : this(numero)
14     {
15         this.limite = limite;
16     }
17 }
```

Tabela 54: Conta.cs

1.18 Exercícios de Fixação

20) Acrescente um construtor na classe Agencia para receber um número como parâmetro.

```
1  class Agencia
2  {
3      public int numero;
4
5      3 references
6      public Agencia(int numero)
7      {
8          this.numero = numero;
9      }
10 }
```

Tabela 55: Agencia cia.cs

21) Tente compilar novamente o arquivo `TestaContaEAgencia`. Observe o erro de compilação.

22) Altere o código da classe `TestaContaEAgencia` para que o erro de compilação seja resolvido.

Substitua linhas **semelhantes** a:

```
1  Agencia a2 = new Agencia();
```

Tabela 56: Código antigo

```
1  Agencia a = new Agencia(123123);
```

Tabela 57: Código novo

Compile novamente o arquivo `TestaContaEAgencia`

23) Acrescente um construtor na classe CartaoDeCredito para receber um número como parâmetro.

```
1  class CartaoDeCredito
2  {
3      public int numero;
4      public string dataDeValidade;
5      public Cliente cliente;
6
7      // Adicione o contrutor abaixo
8
9      3 references
10     public CartaoDeCredito(int numero)
11     {
12         this.numero = numero;
13     }
   }
```

24) Tente compilar novamente os arquivos `TestaCartaoDeCredito` e `TestaClienteECartao`. Observe os erros de compilação.

25) Altere o código das classes `TestaCartaoDeCredito` e `TestaClienteECartao` para que os erros de compilação sejam resolvidos.

Substitua trechos de código **semelhantes** ao trecho abaixo:

```
1  CartaoDeCredito cdc = new CartaoDeCredito();
2  cdc.numero = 11111;
```

Tabela 58: Código antigo

```
1  CartaoDeCredito cdc = new CartaoDeCredito(11111);
```

Tabela 59: Código novo

Compile novamente os arquivos `TestaCartaoDeCredito` e `TestaClienteECartao`.

26) Acrescente um construtor na classe Conta para receber uma referência como parâmetro.

```
1  class Conta
2  {
3      public int numero;
4      public double saldo;
5      public double limite = 100;
6      public Agencia agencia;
7
8      // Adicione o contrutor abaixo
9
10     1 reference
11     public Conta(Agencia agencia)
12     {
13         this.agencia = agencia;
14     }
15
16     1 reference
17     public void Deposita(double valor)
18     {
19         this.saldo += valor;
20     }
21
22     1 reference
23     public void Saca(double valor)
24     {
25         this.saldo -= valor;
26     }
27
28     2 references
29     public void ImprimeExtrato()
30     {
31         System.Console.WriteLine(" SALDO : " + this.saldo);
32     }
33
34     1 reference
35     public double ConsultaSaldoDisponivel()
36     {
37         return this.saldo + this.limite;
38     }
39 }
```

Tabela 60: Código antigo

27) Tente compilar novamente os arquivos `TestaContaEAgencia`, `TestaMetodosConta` e `TestaValoresPadrao`.Observe os erros de compilação.

28) Altere o código das classes `TestaContaEAgencia`, `TestaMetodosConta` e `TestaValoresPadrao` para que os erros de compilação sejam resolvidos.

Substitua trechos de código **semelhantes** ao trecho abaixo:

```
1  Agencia a = new Agencia(1234);
2  Conta c = new Conta();
```

Tabela 61: Código antigo

Por trechos de código semelhantes ao trecho abaixo:

```
1  Agencia a = new Agencia(1234);
2  Conta c = new Conta(a);
```

Tabela 62: Código novo

Também substitua trechos de código semelhantes ao trecho abaixo:

```
1  Conta c = new Conta();
```

Tabela 63: Código antigo

Por trechos de código semelhantes ao trecho abaixo:

```
1  Agencia a = new Agencia(1234);
2  Conta c = new Conta(a);
```

Tabela 64: Código novo

Compile novamente os arquivos `TestaContaEAgencia`, `TestaMetodosConta` e `TestaValoresPadrao`.

1.19 Referências como parâmetro

Da mesma forma que podemos passar valores primitivos como parâmetro para um método ou construtor, também podemos passar valores não primitivos (referências).

Considere um método na classe Conta que implemente a lógica de transferência de valores entre contas. Esse método deve receber como argumento, além do valor a ser transferido, a referência da conta que receberá o dinheiro.

```
1  void Transfere(Conta destino, double valor)
2  {
3      this.saldo -= valor;
4      destino.saldo += valor;
   }
```

Tabela 65: Método Transfere()

Na chamada do método `Transfere()`, devemos ter duas referências de contas: uma para chamar o método e outra para passar como parâmetro.

```
1  Conta origem = new Conta();
2  origem.saldo = 1000;
3
3  Conta destino = new Conta();
4
4  origem.Transfere(destino, 500);
```

Tabela 66: Chamando o método Transfere()

Quando a variável destino é passada como parâmetro, somente a referência armazenada nessa variável é enviada para o método `Transfere()` e não o objeto em si. Em outras palavras, somente o "endereço" para a conta que receberá o valor da transferência é enviado para o método `Transfere().`

1.20 Exercícios de Fixação

29) Acrescente um método na classe Conta para implementar a lógica de transferência de valores entre contas.

```
1  void Transfere(Conta destino, double valor)
2  {
3      this.saldo -= valor;
4      destino.saldo += valor;
   }
```

Tabela 67: Método Transfere()

30) Faça um teste para verificar o funcionamento do método transfere.

```
1  Conta origem = new Conta();
2  origem.saldo = 1000;
3
4  Conta destino = new Conta();
5
6  origem.Transfere(destino, 500);
7
8  Console.WriteLine(origem.saldo);
9  Console.WriteLine(destino.saldo);
```

Tabela 68: TestaMetodoTransfere.cs

Compile e execute a classe `TestaMetodoTransfere`.

1.21 Exercícios Complementares

11) Crie uma pasta chamada complementar dentro da pasta **Orientação_a_Objetos**. Os arquivos a seguir devem ser salvos nessa pasta.

12) Crie uma classe que represente as contas do banco. Essa classe deve conter três atributos: numero, limite e saldo.

13) Crie uma classe chamada TestaConta. Dentro dessa classe, crie um objeto do tipo Conta. Receba do teclado os valores para os atributos numero, saldo e limite. Depois crie um laço que permitaque o usuário escolha a operação que ele deseja realizar. As operações que ele pode realizar são: depositar, sacar e imprimir extrato.

14) Crie uma classe que represente os funcionários do banco. Essa classe deve conter dois atributos: nome e salario. No domínio do banco, obrigatoriamente, os funcionários devem possuir um salário inicial de R$200,00.

15) Crie uma classe chamada TestaFuncionario. Dentro dessa classe, crie um objeto do tipo Funcionario. Receba do teclado o valor para o atributo nome. Depois crie um laço que permita que o usuário possa alterar o nome e o salário dos funcionários e também visualizar os dados atuais.