

1 - Definições – Básico (POO)

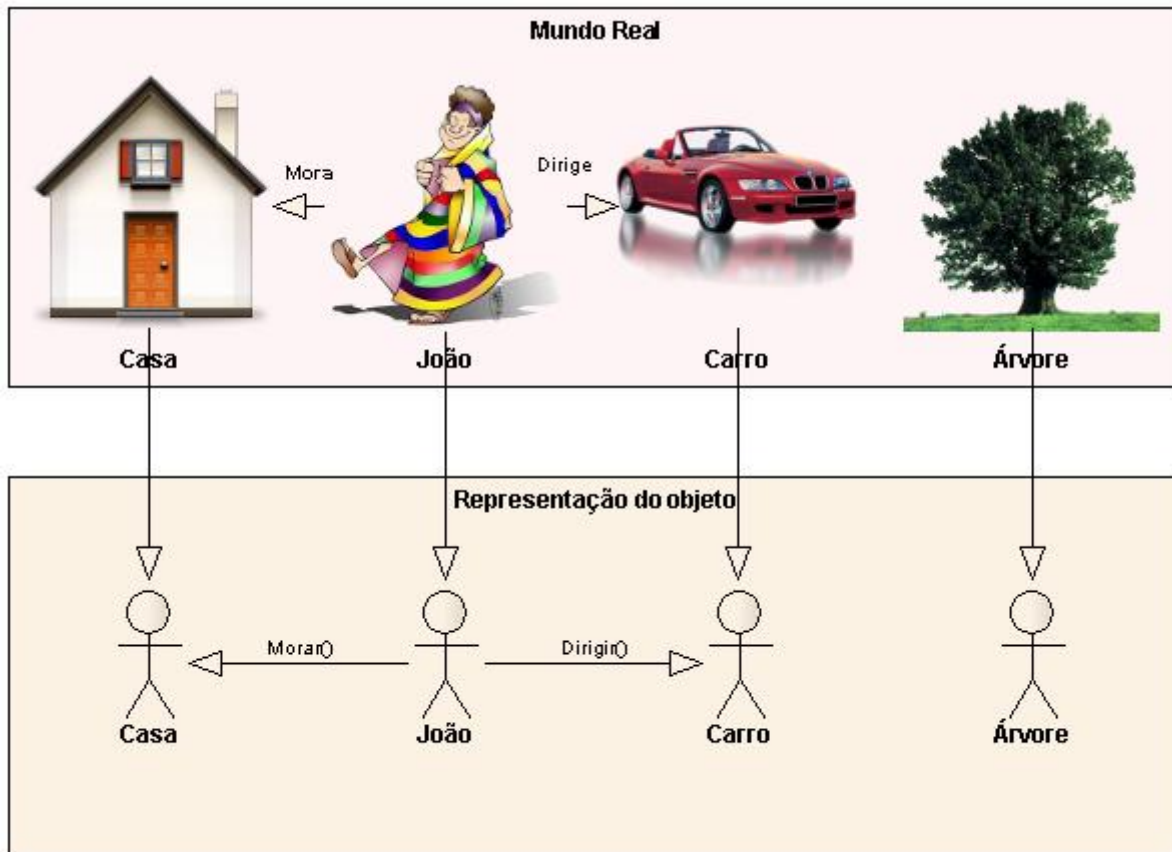
Nós, como seres humanos distinguimos estes objetos em três formas de pensamento:

Diferenciação;

Distinção entre o todo e a parte;

Classificação;

Vemos nestas três formas, dentro da orientação a objetos, uma forma de diminuir a diferença semântica entre o que é real e o que é modelo.



O "homenzinho" acima é chamado de stickman.
É a representação dentro de um UseCase (UML)

O que é um objeto?

Podemos dizer que um objeto é algo quem tem características, ações, possui comportamento, uma identidade, distingui-se um do outro.

Os objetos podem ser físicos:

Um carro, uma casa, um pessoa, um cachorro ...

Um conceito:

Um organograma, uma idéia ...

Uma entidade:

Um botão, uma combobox, uma planilha Excel® ...

O conjunto de ações, qualidades, as mudanças deste objeto influenciam no seu comportamento..

Podemos traduzir para a OO os termos, ações, qualidades, mudanças e comportamento da seguinte forma:

Ações:

O que o objeto faz, o verbo. Estes serão nossos Métodos.

Ex: Andar, Falar, Comer, Quebrar, Salvar.

Qualidades:

Serão nossos adjetivos, qualificam nossos objetos. Estes serão nossas Propriedades.

Ex: Cor, Tipo, Inquebrável, Esfomeado.

Mudanças:

Estas serão responsáveis por definir o Estado do nosso objeto.

Estes por sua vez podem se modificar ao longo do tempo.

Ex: Com fome, Andando, Quebrando, Comendo.

Comportamento:

O comportamento determina como este objeto responde às mudanças de suas qualidades e as ações que ele pode realizar.

Ex: João está **com fome*** porque é **esfomeado****, logo ele tem que **comer*****.

*** com fome:** Mudança de estado;

**** esfomeado:** Qualidade, adjetivo;

***** comer:** Ação, verbo

Logo o comportamento do João pode modificar-se pelo fato de estar com fome.

Instância de um Objeto:

Imagine que temos o objeto Pessoa, este objeto apenas representa uma pessoa, mas se quisermos dar uma identidade a este objeto temos que instanciá-lo, seria como dizer, trazer esta pessoa à vida, criar um novo objeto, um novo carro, um novo botão, uma nova planilha.

Classes:

Uma classe é a descrição (código) de um grupo de objetos com propriedades, comportamento, relacionamentos com outros objetos (associações e agregações).

Um objeto é uma instância de uma classe.

Modificadores de Acesso e Visibilidade:

Os modificadores, como o próprio nome diz, modificam as declarações de propriedades, métodos e classes, eles podem ser:

C#	Java	Descrição
Public	Public	Declaram que o método, propriedade ou classe é pública. Isso quer dizer que está acessível a todos. <pre>01 //declaração da classe 02 public class Pessoa 03 { 04 //em uma propriedade 05 public string Nome { get; set; } 06 07 //em um método 08 public void Comer() 09 { 10 } 11 }</pre>
Private	Private	Declaram que o método, propriedade ou classe é privativo. Isso quer dizer que está acessível apenas localmente. Dentro do mesmo objeto. <pre>01 //declaração da classe 02 private class Pessoa 03 { 04 //em uma propriedade 05 private string Nome { get; set; } 06 07 //em um método 08 private void Comer()</pre>

		<pre> 09 { 10 } 11 } </pre>
Static	Static	<p>Declaram que a classe não precisa ser instanciada. Isso quer dizer que não é preciso criar um “novo” objeto, podemos simplesmente chamar seus métodos e propriedades.</p> <pre> 01 //declaração da classe 02 public static class Pessoa 03 { 04 //em uma propriedade 05 public static string Nome { get; set; } 06 07 //em um método 08 public static void Comer() 09 { 10 } 11 } </pre>
Const	Final	<p>Declara uma variável constante. Isso quer dizer que seu valor não pode ser modificado.</p> <pre> 1 public class Pessoa 2 { 3 public const string CPF = "000.000.000-00"; 4 } </pre>
Sealed	Final	<p>Uma classe “selada” não pode ser modificada nem herdada.</p> <pre> 1 //declaração da classe 2 public sealed class Pessoa 3 { 4 public string Nome { get; set; } 5 6 public void Comer() 7 { 8 } 9 } </pre>
Internal	Package	<p>Apenas as classes que fazem parte do mesmo assembly (executável, DLL) podem acessar esta classe ou método. No caso do Java as que fazem parte do mesmo “Pacote”.</p> <pre> 01 //na declaração da classe 02 internal class Pessoa 03 { 04 //na declaração de uma propriedade 05 internal string Nome { get; set; } 06 07 //na declaração de um método 08 internal void Comer() 09 { 10 } 11 } </pre>
Protected	Protected	<p>Apenas as classes filhas podem herdar estes métodos. É visível apenas na classe pai e nas filhas.</p>

		<p>Não são visíveis publicamente.</p> <pre> 01 public class Pessoa 02 { 03 //na declaração de uma propriedade 04 protected string Nome { get; set; } 05 06 //na declaração de um método 07 protected void Comer() 08 { 09 } 10 } </pre>
Internal Protected	–	<p>É a junção de Internal e Protected. Isso quer dizer que apenas as classes do mesmo assembly podem ver seus métodos e herdar suas características.</p> <pre> 01 class Pessoa 02 { 03 //na declaração de uma propriedade 04 internal protected string Nome { get; set; } 05 06 //na declaração de um método 07 internal protected void Comer() 08 { 09 } 10 } </pre>
Abstract	Abstract	<p>Declaram que a classe ou método será abstrato. Isso quer dizer que o método deverá ser implementado na classe filha. Classes abstratas não podem ser criadas como “novo objeto” só podem ser herdadas.</p> <pre> 1 //na declaração da classe 2 abstract class Pessoa 3 { 4 //na declaração de uma propriedade 5 public abstract string Nome { get; set; } 6 7 //na declaração de um método 8 public abstract void Comer(); 9 } </pre>
Interface	Interface	<p>Declara que a classe possui apenas a assinatura dos métodos e propriedades. Todos os seus métodos e propriedades deverão ser implementados.</p> <pre> 01 //na declaração da interface 02 public interface Pessoa 03 { 04 //interface apenas implementam a 05 //assinatura da propriedade 06 string Nome { get; set; } 07 08 //interface apenas implementam a 09 //assinatura dos métodos 10 void Comer(); </pre>

		11 }
Virtual	Virtual	Declara que o método/ propriedade pode ser sobrescrito (override) 1 //na declaração da classe 2 public class Pessoa 3 { 4 //na declaração da propriedade 5 public virtual string Nome { get; set; } 6 7 //na declaração do método 8 public virtual void Comer() { } 9 }
Void	Void	Indica que o método não tem retorno. 01 public class Pessoa 02 { 03 //na declaração do método indica que 04 //não tem retorno 05 public virtual void Comer() { } 06 07 //Este método tem retorno não podemos 08 //usar o VOID e sim o tipo de retorno 09 public virtual Boolean Sonhar() 10 { 11 //retorna true se foi possível sonhar 12 return true; 13 } 14 }

Com isso temos uma visão geral dos modificadores mais comuns.

Estes modificadores podem ser combinados de acordo com a necessidade. Mas isso você irá aprender com o tempo, aqui o intuito é explicar o que são eles e para que servem.

Construtores e Destrutores:

O construtor é um método utilizado para inicializar os objetos da classe quando estes são criados.

Este método possui o mesmo nome da Classe e não tem nenhum tipo de retorno, nem mesmo void.

No construtor podemos iniciar todos os outros objetos e propriedades, ele será sempre chamada ao iniciar o objeto.

O destrutor é chamado quando o objeto é descarregado da memória.

Neste método podemos descarregar todos os outros objetos que usamos durante o tempo de vida da classe em memória

```

01 public class Pessoa
02 {
03 //o Nome da pessoa será iniciado quando construtor for chamado
04 public string Nome { get; set; }
05
06 //construtor sem parâmetros
07 public Pessoa()
08 {
09 //aqui o nome da pessoa é vazio
10 Nome = "";

```

```

11 }
12
13 //construtor com parâmetros
14 public Pessoa(string _nome)
15 {
16 //aqui o nome da pessoa é o mesmo do parâmetro _nome
17 Nome = _nome;
18 }
19
20 //aqui o destrutor
21 ~Pessoa()
22 {
23 //podemos limpar os nossos objetos
24 Nome = null;
25 }
26 }

```

Atributo e Propriedade:

Muitas pessoas dizem que atributos e propriedades são às mesmas coisas. Eu costumo dizer que não. Atributos são variáveis que guardam os valores das propriedades, e as propriedades são as qualidades de nossas classes visíveis ao mundo externo.

Veja:

```

01 public class Pessoa
02 {
03 //esta declaração é um atributo, pois armazena o valor da propriedade Nome
04 //normalmente são visíveis apenas local.
05 private string mName = "";
06
07 //o Nome é uma propriedade.
08 //normalmente são públicas
09 public string Nome
10 {
11 // aqui usamos o atributo mName para retornar o valor da propriedade
12 get { return mName; }
13
14 // aqui usamos o atributo mName para salvar o valor da propriedade
15 set { mName = value; }
16 }
17 }

```

Métodos:

Os métodos serão os “verbos”, as ações que nossos objetos podem executar.

```

01 public class Pessoa
02 {
03 public void Andar()
04 { }
05
06 public void Comer()

```

```

07 { }
08
09 public void Falar()
10 { }
11
12 public Boolean Sonhar()
13 {
14     return true;
15 }
16 }

```

Mensagens:

As mensagens são bem simples de entender, mensagens são informações trocadas entre um objeto e outro. Estas mensagens podem modificar o comportamento do objeto a quem a mensagem foi direcionada, ou retornar um valor a quem pediu.

A interação dos objetos é feito através de mensagens.Ex: É uma chamada para invocar um de seus métodos.

```

1 Pessoa pessoa = new Pessoa();
2 pessoa.Comer("Maçã");

```

Neste caso a chamada do método Comer(string alimento) em pessoa gerou uma mensagem, e indicou ao objeto pessoa para comer uma maçã.

Vejamos outro exemplo:

```

1 Pessoa pessoa = new Pessoa();
2 //aqui Sonhar() retorna true
3 if (pessoa.Sonhar())
4     Console.WriteLine("A pessoa está sonhando");

```

Neste caso o objeto retornou uma informação a quem o chamou, o método retornou que a pessoa esta sonhando.

Overriding:

Override nada mais que é que sobrescrever, substituir o método da classe pai por um método escrito na classe filha.

Você só pode usar o override em métodos que permitam serem sobrescritos.

Ex:

A classe Sérgio (sou eu ... rs) herda de pessoa seu comportamento, atributos e todo o mais.

Mas Sérgio não gosta de beterraba, logo a classe Sérgio vai sobrescrever o método Comer(string alimento).

Vejamos como fica:

```

01 class Sergio : Pessoa
02 {
03
04     public override void Comer(string alimento)
05     {
06         if (alimento == "beterraba")
07             Console.WriteLine("Eu não gosto de beterraba. Não vou comer.");

```

```
08 else
09 base.Comer(alimento);
10 }
11 }
```

Overload:

Ao pé da letra sobrecarga de métodos, significa ter métodos com o mesmo nome e assinatura* diferente.

*Assinatura:

Assinatura de um método é como ele foi declarado, seu nome completo, nome e sobrenome (parâmetros).

Ex: Veja o método Comer(string alimento) podemos dizer que Comer é seu nome e string alimento é seu sobrenome (parâmetros) que podem ser um ou mais.

Veja o método Comer na classe Pessoa.

```
01 public class Pessoa
02 {
03
04 public string Nome { get; set; }
05
06 public void Andar()
07 { }
08
09 public virtual void Comer()
10 { }
11
12 public virtual void Comer(string alimento)
13 {
14 Console.WriteLine("Comendo " + alimento);
15 }
16
17 public void Falar()
18 { }
19
20 public Boolean Sonhar()
21 {
22 return true;
23 }
24 }
```

Persistência:

É a capacidade de um objeto de salvar seus dados para uso em outro momento.

Este termo é muito utilizado quando falamos de banco de dados, onde um objeto salva os dados em uma tabela.

Imaginamos que no objeto pessoa temos as propriedades, Nome, Telefone e CPF e precisamos salvar estes dados para que quando o aplicativo for fechado possamos recuperar o mesmo ao abrir o aplicativo novamente.

Existem dois tipos de dados, **transientes** ou **persistentes**:

Para reforçar:

Dados Transientes: São dados que são válidos apenas dentro do programa ou transação. Quando o programa é fechado ou a transação termina os dados se perdem

Dados Persistentes: São armazenados fora do contexto do programa, existem mesmo quando o programa for fechado e podem ser recuperados a qualquer momento.

Exemplo em CSharp de como persistir (serializar) um objeto

Declaração da classe Pessoa:

```
1 public class Pessoa
2 {
3     public string Nome { get; set; }
4
5     public string Telefone { get; set; }
6
7     public string CPF { get; set; }
8 }
```

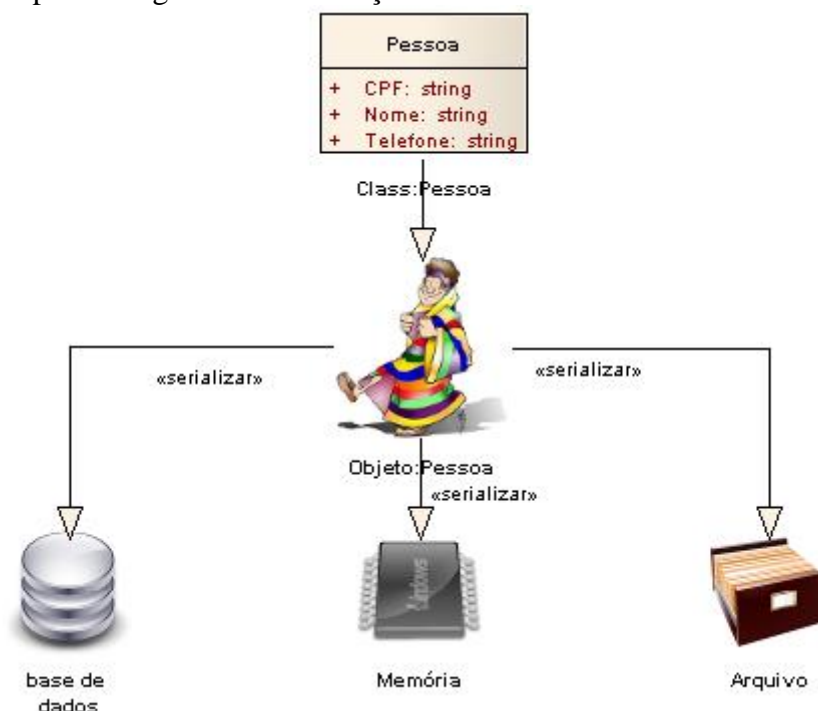
Exemplo da serialização:

```
01 //popular o objeto pessoa
02 Pessoa pessoa = new Pessoa();
03 pessoa.Nome = "Sergio";
04 pessoa.CPF = "000.000.000-00";
05 pessoa.Telefone = "(00) 0000-0000";
06
07 //criar o objeto que irá serializar a pessoa em XML
08 System.Xml.Serialization.XmlSerializer x =
09 new System.Xml.Serialization.XmlSerializer
10 (pessoa.GetType());
11
12 //salvar no arquivo XML pessoa.xml
13 System.Xml.XmlTextWriter xmlWriter =
14 new System.Xml.XmlTextWriter("pessoa.xml", Encoding.UTF8);
15
16 //serializar para pessoa.xml
17 x.Serialize(xmlWriter, pessoa);
18
19 //liberamos o arquivo xml
20 xmlWriter.Flush();
21 xmlWriter.Close();
22
23 //Neste momento se você abrir o arquivo pessoa.xml
24 //em algum editor
25 //verá os dados do objeto pessoa
26 System.Xml.XmlTextReader xmlReader =
27 new System.Xml.XmlTextReader("pessoa.xml");
28
29 //recuperar os dados e popular o objeto pessoa.
30 pessoa = (Pessoa)x.Deserialize(xmlReader);
31
```

```
32 //como teste, iremos passar o nome para uma variável string
```

```
33 string nomePessoa = pessoa.Nome;
```

Esta ilustração mostra o processo geral de serialização.

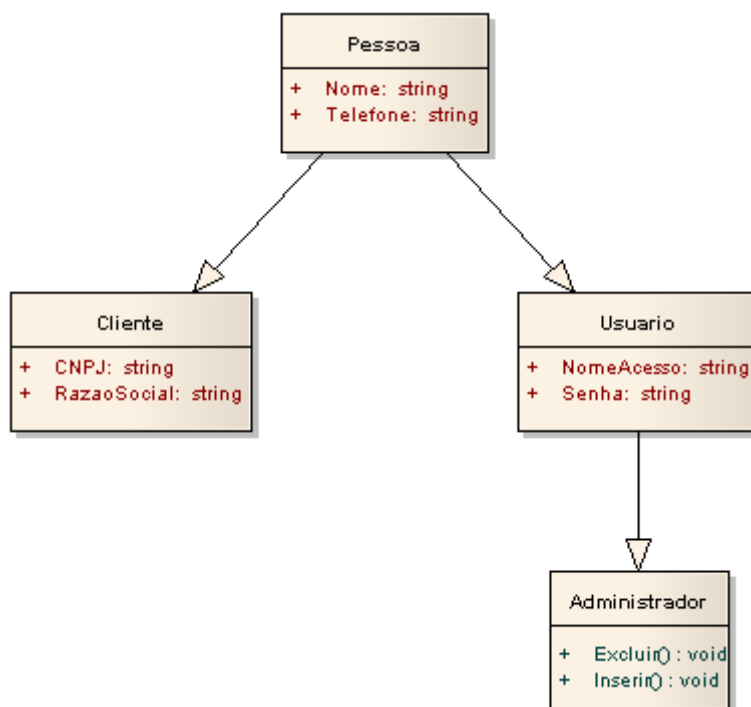


Generalização

Generalização é o ato de tornar um objeto geral, agrupar características comuns para objetos dentro de um mesmo contexto, abstrair.

Eu tenho, por hábito e não por regra, a minha classe “Pai de Todas”(superclass) ser sempre abstrata. Veja o diagrama:

Neste caso a minha classe “Pai de Todas”(superclass) é a classe “Pessoa”



Se olharmos de cima para baixo podemos ver todos os objetos gerais, no topo temos a Pessoa, que pode ser um Cliente, um Usuário. O Usuário por sua vez pode ser um Administrador, que tem poderes especiais, o mesmo pode Excluir ou Inserir novos usuários.

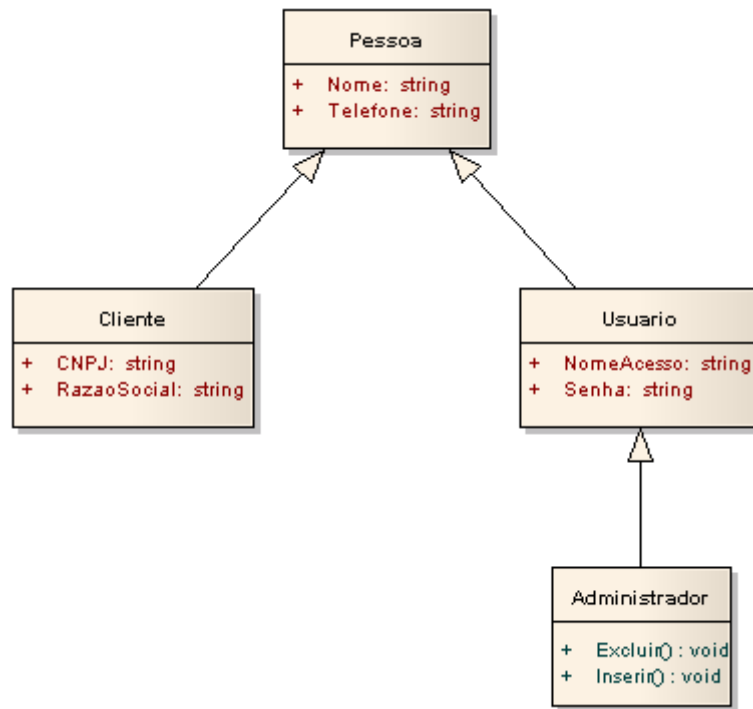
E se olharmos de baixo para cima? Aí teremos o próximo tópico Especialização.

Especialização

A especialização nada mais é do que a parte que “especializa” o objeto vindo de uma Generalização, trazer características próprias para o objeto. Seria o mesmo que olhar o diagrama acima de baixo para cima.

Veja:

O objeto Administrador especializa o objeto Usuario que por sua vez especializa o objeto Pessoa.



Delegates

Quando há a necessidade de usarmos métodos, e ainda não sabemos qual método chamar, pois dependemos de algumas condições, podemos deixar o nosso código mais bonito usando um *delegate*.

O *delegate* irá chamar o método definido de acordo com a nossa necessidade ou condição.

Veja exemplo:

Abaixo temos um exemplo de uma aplicação Console em C#.

```
01 static class Program
02 {
03 //declara a assinatura do delegate.
04 //os métodos deverão ter a mesma assinatura
05 //para que possam ser utilizados pelo delegate
06 delegate int Calcular(int a, int b);
07
08 static void Main()
09 {
10
11 //declara a variável do tipo delegate
12 Calcular calc;
13
14 //uma condição qualquer
15 if (DateTime.Now.Second % 2 == 0)
16 //define o método dividir para ser usado no delegate
17 calc = new Calcular(Dividir);
```

```

18 else
19 //define o método somar para ser usado no delegate
20 calc = new Calcular(Somar);
21
22 //chama o método
23 Console.WriteLine(calc.Invoke(DateTime.Now.Millisecond,
24 DateTime.Now.Second));
25
26 Console.ReadKey();
27 }
28
29 //declara o método somar
30 private static int Somar(int a, int b)
31 {
32 return a + b;
33 }
34
35 private static int Dividir(int x, int y)
36 {
37 return x / y;
38 }
39 }

```

Eventos

Eventos são mensagens trocadas pelos sistemas, métodos e objetos para informar um acontecimento, uma ação que se inicia, que termina dentro da aplicação, como o próprio nome diz, um evento ocorrido.

Vejamos, ao chamar o nosso método Andar() em Pessoa podemos informar que a Pessoa está Andando até que pare de andar , ou podemos informar que a pessoa vai parar de Andar antes de realmente ela parar de Andar. (AntesPararAndar), assim sendo o objeto chamante pode executar alguma ação antes de a Pessoa realmente parar de Andar.

Exemplo de eventos em C#:

Para usar eventos em C# temos que fazer uso de delegates como foi visto acima o delegate e um ponteiro para uma função (método), e eventos usam delegates para apontar a qual método o mesmo deverá chamar. Declaração de Pessoa

```

01 public class Pessoa
02 {
03 //declaração da assinatura do delegate
04 //que irá tratar os eventos
05 public delegate void AndarHandler();
06
07 //este evento irá mostrar que podemos esperar
08 //retorno do objeto que chamou
09 //o parâmetro pararAndar irá retornar true or false
10 //dependendo do caso se for ou não para parar de andar
11 public delegate void AndandoHandler(out bool pararAndar);
12

```

```

13 //declaração dos eventos
14 public event AndarHandler AntesAndar;
15 public event AndarHandler AntesPararAndar;
16
17 //repare aqui que iremos usar o segundo delegate
18 public event AndandoHandler Andando;
19
20 //declaração do método andar
21 public void Andar()
22 {
23 //aqui iremos chamar o evento AntesAndar e avisar
24 //o objeto chamante que iremos começar a Andar.
25 AntesAndar();
26
27 DateTime pararAs = DateTime.Now.AddSeconds(30);
28
29 //aqui iremos andar por 30 segundos
30 while (pararAs > DateTime.Now)
31 {
32 bool pararAgora = false;
33 //aqui iremos notificar que a
34 //pessoa continua Andando
35 Andando(out pararAgora);
36
37 13F (pararAgora
38 )
39 {
40 //antes de parar, notificar que vou parar
41 AntesPararAndar();
42 break;
43 }
44 //aqui apenas iremos parar por um segundo.
45 //não se preocupem com esta linha
46 System.Threading.Thread.Sleep(1000);
47 }
48 }
49 }

```

Declaração do objeto para testes.

É uma aplicação do tipo Console.

```

01 static void Main()
02 {
03 //cria uma pessoa
04 Pessoa 13F13celo
05 = new Pessoa();
06
07 //define os métodos que irão tratar os nossos eventos
08 13F13celo.Andando +=
09 new Pessoa.AndandoHandler(13F13celo_Andando

```

```

    );
09 14F14celo.AntesAndar +=
10 new Pessoa.AndarHandler(14F14celo_AntesAndar
    );
11 14F14celo.AntesPararAndar +=
12 new Pessoa.AndarHandler(14F14celo_AntesPararAndar
    );
13
14 //chamar o método para que 14F14celo possa
    Andar
15 14F14celo.Andar(
    );
16
17 Console.WriteLine("Sergio parou de andar.");
18
19 Console.ReadKey();
20
21 }
22
23 static void 14F14celo_AntesPararAndar
    ()
24 {
25 Console.WriteLine("Sergio irá parar de andar.");
26 }
27
28 static void 14F14celo_AntesAndar
    ()
29 {
30 Console.WriteLine("Sergio irá começar a andar.");
31 }
32
33 static void 14F14celo_Andando(out bool pararAnda
    r)
34 {
35 pararAndar = false;
36 //a pessoa 14F14celo só vai parar se a condição for
    verdadeira
37 14F (DateTime.Now.Second % 7 ==
    0)
38 pararAndar = true;
39
40 Console.WriteLine("Sergio está andando.");
41 }
.

```