



Programação e Sistemas de Informação

CURSO PROFISSIONAL TÉCNICO DE
GESTÃO E PROGRAMAÇÃO DE SISTEMAS INFORMÁTICOS

Estruturas de Dados Dinâmicas

MÓDULO 6

Professor: João Martiniano

Introdução

- A .NET Framework disponibiliza várias estruturas de dados
- Diferentes problemas exigem diferentes formas de armazenar os dados
- É assim, importante saber que:
 - existem várias estruturas de dados
 - cada uma tem determinadas características (performance e quantidade de memória que consome), adequadas para utilizar em determinadas situações

Introdução

- Neste módulo serão abordados os seguintes conteúdos:
 - a estrutura `ArrayList`
 - *generics*
 - a estrutura `Queue`
 - a estrutura `Stack`
 - a estrutura `List`
 - a estrutura `Dictionary`

Introdução

- Até este ponto, a única forma que analisámos, para armazenar conjuntos de dados, foram os arrays
- No entanto os arrays têm limitações:
 - são pouco flexíveis
 - para determinadas operações são lentos
 - são pouco sofisticados
- Em suma, são adequados apenas em determinadas situações
- Quando se pretendem escrever programas com requisitos um pouco mais avançados, devem ser utilizadas outras estruturas de dados, nomeadamente **coleções**

As coleções

- Dados que são semelhantes (têm uma estrutura similar) são armazenados de forma mais eficiente em coleções
- Existem dois tipos de coleções:
 - genéricas
 - não genéricas
- Algumas características comuns das coleções:
 - é possível enumerar a coleção
 - é possível copiar os conteúdos das coleções para um array

As coleções

- Guia simplificado para escolher uma estrutura de dados:

Objetivo	Coleção Genérica	Coleção não-genérica
Aceder aos itens através de um índice	List<T>	Array ArrayList
Obter itens <i>First In First Out</i> (primeiro a entrar, primeiro a sair)	Queue<T>	Queue
Usar itens <i>Last In First Out</i> (último a entrar, primeiro a sair)	Stack<T>	Stack
Aceder sequencialmente a itens	LinkedList<T>	Queue Stack ArrayList
Armazenar itens no formato chave/valor; acesso rápido pela chave	Dictionary<TKey, TValue>	Hashtable

(adaptado de <https://docs.microsoft.com/en-us/dotnet/standard/collections/selecting-a-collection-class>)

- Importa salientar que existem muitas mais coleções (genéricas e não genéricas), na .NET Framework

|ArrayList

ArrayList

- Um `ArrayList` é uma coleção semelhante a um array
- Tal como num array, os elementos são acedidos através de um índice
- Características:
 - pode acomodar um grande número de elementos
 - a capacidade de um `ArrayList` é a quantidade de elementos que pode conter
 - ou seja, pode ter uma capacidade maior do que os elementos que contém num determinado momento
 - qualquer elemento pode ser inserido ou removido
 - podem existir elementos duplicados
 - é necessário importar o namespace `System.Collections`
- Os dados são convertidos e armazenados com o tipo `Object`
- Consequentemente, há uma penalização em termos de performance
- Por esta razão **já não é aconselhável utilizar a coleção `ArrayList`**: deve ser utilizada a coleção genérica `List<T>`

ArrayList

Inicialização

- Exemplo:

```
ArrayList listaCidades = new ArrayList();

listaCidades.Add("Lisboa");
listaCidades.Add("Coimbra");
listaCidades.Add("Viana do Castelo");
listaCidades.Add("Faro");

// Mostrar o primeiro elemento ("Lisboa")
Console.WriteLine(listaCidades[0]);

// Obter o número de elementos
Console.WriteLine($"\\nO ArrayList possui {listaCidades.Count} elementos");

// Percorrer a lista
Console.WriteLine("\\nConteúdo da lista:");
foreach (Object cidade in listaCidades)
{
    Console.WriteLine(cidade);
}
```

ArrayList

- Para inserir elementos:

```
// Inserir um elemento após o 1º elemento  
listaCidades.Insert(1, "Porto");  
  
// Inserir um elemento no final  
listaCidades.Insert(listaCidades.Count, "Évora");  
  
// Após as operações anteriores, o ArrayList fica com os seguintes elementos:  
// "Lisboa", "Porto", "Coimbra", "Viana do Castelo", "Faro", "Évora"
```

- Para remover elementos:

```
// Remover o 4º elemento ("Viana do Castelo")  
// (não esquecer que o 4º elemento tem índice 3)  
listaCidades.RemoveAt(3);  
  
// Após a operação anterior, o ArrayList fica com os seguintes elementos:  
// "Lisboa", "Porto", "Coimbra", "Faro", "Évora"
```

| *Generics*

Generics

- Tradicionalmente, as estruturas de dados são criadas para um determinado tipo de dados
- Até ao aparecimento da tecnologia de *generics* não era possível utilizar estruturas de dados com qualquer tipo de dados
- Pelo contrário, com *generics* é possível ter estruturas de dados que acomodam qualquer tipo de dados
- Ou seja:
 - uma estrutura *generic* é uma estrutura genérica
 - implementa um conjunto de operações (inserir, editar, eliminar, ordenar, etc.) que funcionam com qualquer tipo de dados

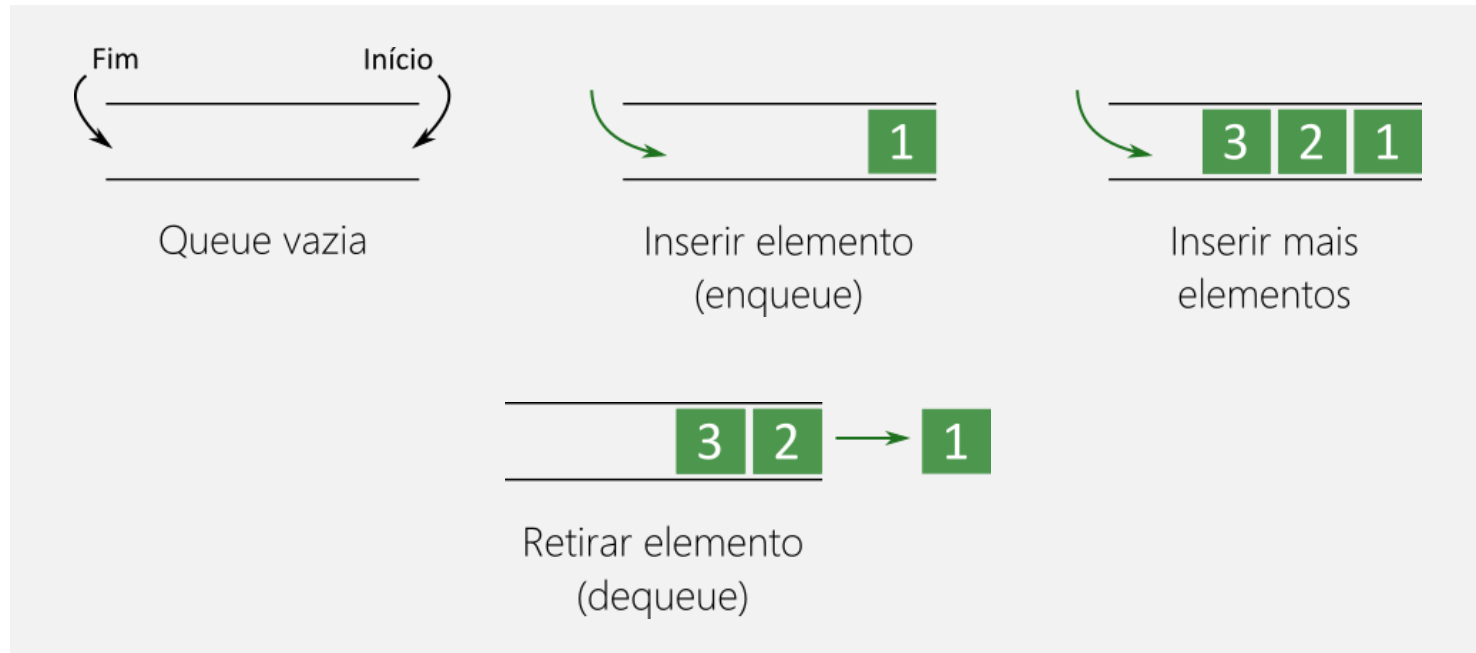
| *Queue*

Queue

- **Queue** = fila
- É uma estrutura **FIFO** (*First In First Out*):
 - o primeiro elemento a entrar é o primeiro elemento a sair
- Ou seja, funciona tal como uma fila de espera na vida real:
 - o primeiro elemento é colocado no início da fila
 - os elementos seguintes são colocados atrás do primeiro elemento
 - os elementos são retirados a partir do início da fila

Queue

- As operações básicas de uma *queue* são:
 - inserir um elemento na *queue*: operação **enqueue**
 - retirar um elemento da *queue*: operação **dequeue**



Queue: Declaração

- Para criar uma nova *queue* utiliza-se a seguinte sintaxe:

```
Queue<tipo> nome = new Queue<tipo>();
```

- Em que:

tipo O tipo de dados que a *queue* irá armazenar (*int*, *string*, etc.)

nome O nome da *queue*

- Exemplo:

```
Queue<string> clientes = new Queue<string>();
```

Tipo

Nome

- Neste exemplo a *queue* *clientes* armazena dados do tipo *string* (o nome de clientes)

Queue: Inserção de elementos

- Para inserir elementos numa *queue* é utilizado o método `Enqueue()`
- Este método insere um elemento no fim da *queue*

- Sintaxe:

`Enqueue(elemento)`

- Em que:

`elemento` O elemento a inserir

- Exemplo:

```
Queue<string> clientes = new Queue<string>();  
  
clientes.Enqueue("Paula");  
clientes.Enqueue("José Ramirez");  
clientes.Enqueue("Alberto");  
clientes.Enqueue("Catarina");
```

Queue

Exercício

- Complete o seguinte código para:
 - declarar uma *queue* do tipo `int`
 - adicionar 2 elementos à *queue* (os números 300 e 5)

```
Queue<_____> numeros = new Queue<_____>();  
  
_____._____ (300);  
_____._____ (5);
```

Queue

Exercício: Resolução

- Complete o seguinte código para:
 - declarar uma *queue* do tipo `int`
 - adicionar 2 elementos à *queue* (os números 300 e 5)

```
Queue<int> numeros = new Queue<int>();  
  
numeros.Enqueue(300);  
numeros.Enqueue(5);
```

Queue: Remoção de elementos

- Para remover elementos de uma *queue* é utilizado o método `Dequeue()`
- Este método remove e retorna o elemento que se encontra no **início** da *queue*

- Sintaxe:

`Dequeue()`

- Exemplo:

```
Queue<string> clientes = new Queue<string>();

clientes.Enqueue("Paula");
clientes.Enqueue("José Ramirez");
clientes.Enqueue("Alberto");
clientes.Enqueue("Catarina");

// Retirar o primeiro elemento e colocá-lo na variável cliente
string cliente = clientes.Dequeue();
Console.WriteLine($"Foi retirado o cliente {cliente} da queue");

// A partir deste momento a queue contém os seguintes elementos:
//   "José Ramirez", "Alberto", "Catarina"
```

Queue: O método `Peek()`

- O método `Peek()` permite obter o elemento que se encontra no início da *queue* mas sem o retirar

```
Queue<string> clientes = new Queue<string>();

clientes.Enqueue("Paula");
clientes.Enqueue("José Ramirez");
clientes.Enqueue("Alberto");
clientes.Enqueue("Catarina");

// Ver o elemento que se encontra no início da queue ("Paula")
string primeiroCliente = clientes.Peek();
Console.WriteLine($"Cliente no início da queue: {primeiroCliente}");
```

Queue: Percorrer os elementos

- Para percorrer uma *queue* é utilizada a instrução **foreach**:

```
Queue<string> clientes = new Queue<string>();

clientes.Enqueue("Paula");
clientes.Enqueue("José Ramirez");
clientes.Enqueue("Alberto");
clientes.Enqueue("Catarina");

// Percorrer a queue
Console.WriteLine("Conteúdo da queue:");
foreach (string c in clientes)
{
    Console.WriteLine(c);
}
```

Queue: Obter o número de elementos

- A propriedade `Count` devolve o número de elementos numa *queue*

```
Queue<string> clientes = new Queue<string>();

clientes.Enqueue("Paula");
clientes.Enqueue("José Ramirez");
clientes.Enqueue("Alberto");
clientes.Enqueue("Catarina");

// Número de elementos na queue
Console.WriteLine($"A queue possui {clientes.Count} elementos");
```

Queue: Limpar o conteúdo

- Por vezes é necessário limpar uma *queue*
- Nesses casos é utilizado o método `Clear()`: este método remove todos os elementos

```
Queue<string> clientes = new Queue<string>();

clientes.Enqueue("Paula");
clientes.Enqueue("José Ramirez");
clientes.Enqueue("Alberto");
clientes.Enqueue("Catarina");

// Número de elementos na queue
Console.WriteLine($"A queue possui {clientes.Count} elementos");

// Limpar a queue
clientes.Clear();

// Número de elementos na queue (0 elementos)
Console.WriteLine($"A queue possui {clientes.Count} elementos");
```

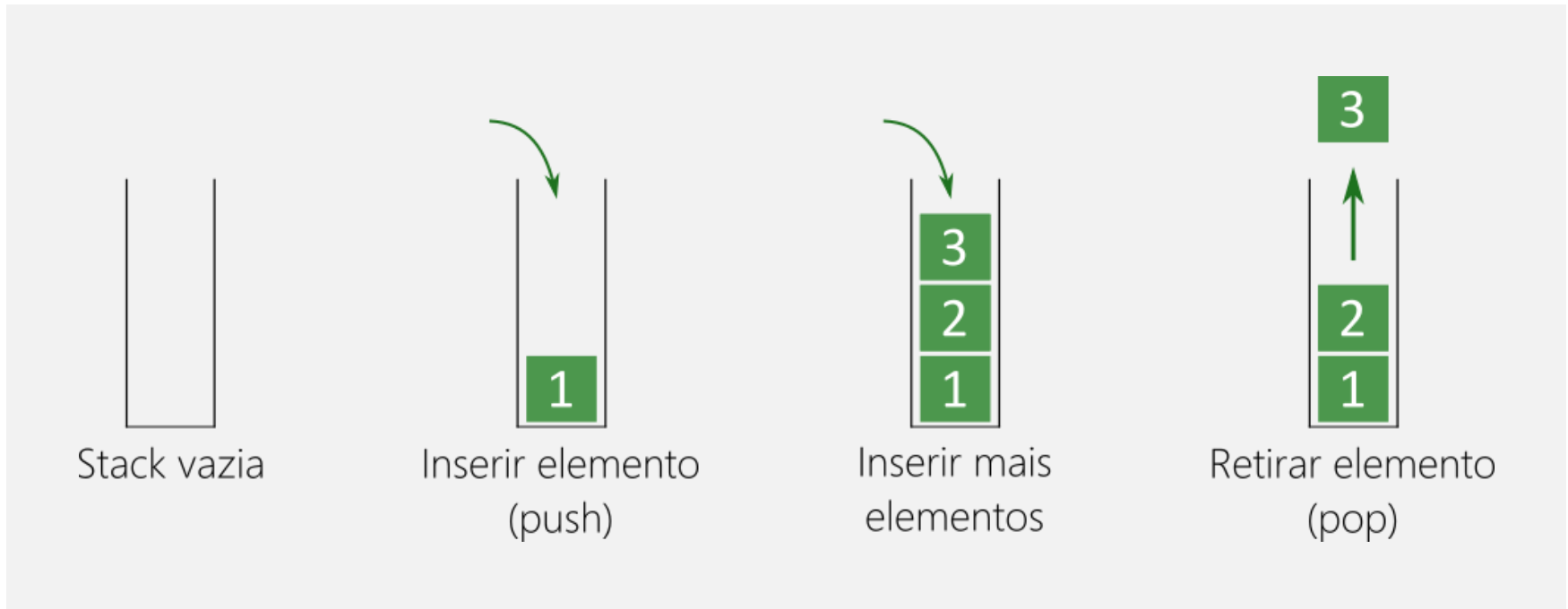

| *Stack*

Stack

- **Stack** = pilha
- É uma estrutura **LIFO** (*Last In First Out*):
 - o último elemento a entrar é o primeiro elemento a sair
- Ou seja, funciona tal como uma pilha de objetos (livros, por exemplo) na vida real:
 - o primeiro elemento é colocado no fundo da pilha
 - os elementos seguintes são colocados por cima do primeiro elemento
 - os elementos são retirados a partir do topo da pilha
 - o primeiro elemento a ter sido colocado, é o último a ser retirado

Stack

- As operações básicas de uma *stack* são:
 - inserir um elemento: operação **push**
 - retirar um elemento: operação **pop**



Stack: Declaração

- Para criar uma nova *stack* utiliza-se a seguinte sintaxe:

```
Stack<tipo> nome = new Stack<tipo>();
```

- Em que:

tipo O tipo de dados que a *stack* irá armazenar (*int*, *string*, etc.)

nome O nome da *stack*

- Exemplo:

```
Stack<double> temperaturas = new Stack<double>();
```

Tipo

Nome

- Neste exemplo a *stack* *temperaturas* armazena dados do tipo *double*

Stack: Inserção de elementos

- Para inserir elementos numa *stack* é utilizado o método `Push()`

- Sintaxe:

`Push(elemento)`

- Em que:

`elemento` O elemento a inserir

- Exemplo:

```
Stack<double> temperaturas = new Stack<double>();  
  
temperaturas.Push(-10.8D);  
temperaturas.Push(9.0D);  
temperaturas.Push(22.3D);  
temperaturas.Push(1.2D);
```

Stack

Exercício

- Complete o seguinte código para:
 - declarar uma *stack* do tipo `string`
 - adicionar 2 elementos à *stack* ("Teste1.pdf" e "RelatórioDespesas.xlsx")

```
Stack<_____> documentos = _____ Stack<_____>();  
  
_____._____("Teste1.pdf");  
_____._____("RelatórioDespesas.xlsx");
```

Stack

Exercício: Resolução

- Complete o seguinte código para:
 - declarar uma *stack* do tipo `string`
 - adicionar 2 elementos à *stack* ("Teste1.pdf" e "RelatórioDespesas.xlsx")

```
Stack<string> documentos = new Stack<string>();  
  
documentos.Push("Teste1.pdf");  
documentos.Push("RelatórioDespesas.xlsx");
```

Stack: Remoção de elementos

- Para remover elementos de uma *stack* é utilizado o método `Pop()`
- Este método remove e retorna o elemento que se encontra no **topo** da *stack*

- Sintaxe:

`Pop()`

- Exemplo:

```
Stack<double> temperaturas = new Stack<double>();

temperaturas.Push(-10.8D);
temperaturas.Push(9.0D);
temperaturas.Push(22.3D);
temperaturas.Push(1.2D);

// Retirar o último elemento e colocá-lo na variável temperatura
double temperatura = temperaturas.Pop();
Console.WriteLine($"Foi retirada a temperatura {temperatura} da stack");

// A partir deste momento a stack contém os seguintes elementos:
// 22.3, 9.0, -10.8
```


Stack: O método `Peek()`

- O método `Peek()` permite obter o elemento que se encontra no topo da *stack* mas sem o retirar

```
Stack<double> temperaturas = new Stack<double>();

temperaturas.Push(-10.8D);
temperaturas.Push(9.0D);
temperaturas.Push(22.3D);
temperaturas.Push(1.2D);

// Ver o elemento que se encontra no topo da stack (1.2)
double primeiraTemperatura = temperaturas.Peek();
Console.WriteLine($"Temperatura no início da stack: {primeiraTemperatura}");
```

Stack: Percorrer os elementos

- Para percorrer uma *stack* é utilizada a instrução `foreach`:

```
Stack<double> temperaturas = new Stack<double>();

temperaturas.Push(-10.8D);
temperaturas.Push(9.0D);
temperaturas.Push(22.3D);
temperaturas.Push(1.2D);

// Percorrer a stack
Console.WriteLine("Conteúdo da stack:");
foreach (double t in temperaturas)
{
    Console.WriteLine(t);
}
```

Stack: Obter o número de elementos

- A propriedade `Count` devolve o número de elementos numa *stack*

```
Stack<double> temperaturas = new Stack<double>();  
  
temperaturas.Push(-10.8D);  
temperaturas.Push(9.0D);  
temperaturas.Push(22.3D);  
temperaturas.Push(1.2D);  
  
// Número de elementos na stack  
Console.WriteLine($"A stack possui {temperaturas.Count} elementos");
```

Stack: Limpar o conteúdo

- Por vezes é necessário limpar uma *stack*
- Nesses casos é utilizado o método `Clear()`: este método remove todos os elementos

```
Stack<double> temperaturas = new Stack<double>();

temperaturas.Push(-10.8D);
temperaturas.Push(9.0D);
temperaturas.Push(22.3D);
temperaturas.Push(1.2D);

// Número de elementos na stack
Console.WriteLine($"A stack possui {temperaturas.Count} elementos");

// Limpar a stack
temperaturas.Clear();

// Número de elementos na stack (0 elementos)
Console.WriteLine($"A stack possui {temperaturas.Count} elementos");
```

| *List*

List

- **List = lista**
- Numa *list* os dados são acedidos através de um índice numérico, tal como num array
- Esta estrutura de dados é, no entanto, muito mais flexível que um array
- Podem ser acrescentados ou eliminados dados de forma fácil

List: Declaração

- Para criar uma nova *list* utiliza-se a seguinte sintaxe:

```
List<tipo> nome = new List<tipo>()
```

- Em que:

tipo O tipo de dados que a *list* irá armazenar (*int*, *string*, etc.)

nome O nome da *list*

List: Declaração

- Exemplo:

```
public struct Cliente
{
    public string nome;
    public int telemovel;

    public Cliente(string _nome, int _telemovel)
    {
        nome = _nome;
        telemovel = _telemovel;
    }
}
```

```
List<Cliente> clientes = new List<Cliente>();
```

Tipo

Nome

- Neste exemplo a *list* `clientes` armazena dados do tipo `Cliente`

List: Inserção de elementos

- Existem vários métodos para inserir elementos numa *list* entre os quais:
 - o método `Add()`: adiciona um elemento ao final da *list*
 - o método `Insert()`: insere um elemento num índice específico da *list*

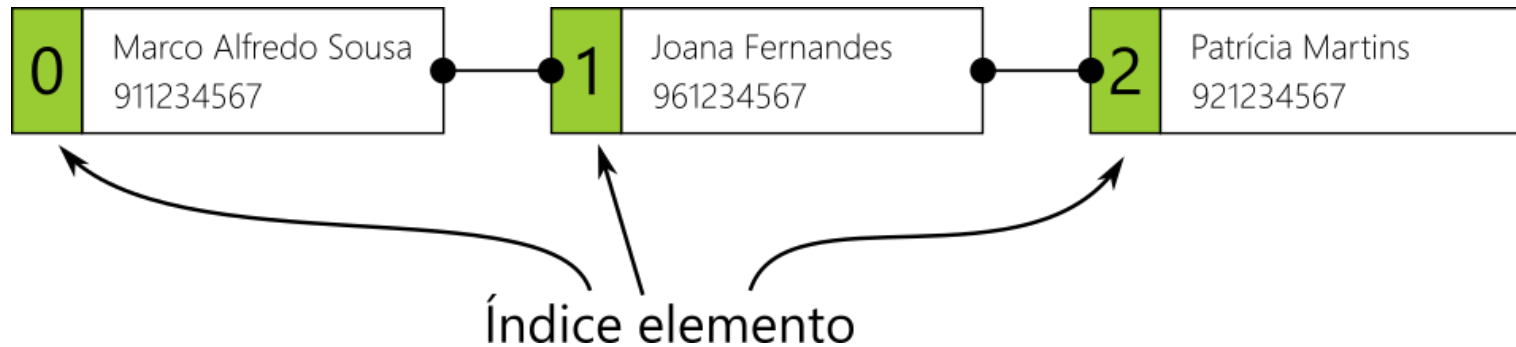
List: Inserção de elementos com o método `Add()`

- O método `Add()` adiciona um elemento ao final da *list*
- Sintaxe:
`Add(elemento)`
- Em que:
`elemento` O elemento a inserir
- Exemplo:

```
List<Cliente> clientes = new List<Cliente>();  
  
// Adicionar clientes  
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));  
clientes.Add(new Cliente("Joana Fernandes", 961234567));  
clientes.Add(new Cliente("Patrícia Martins", 921234567));
```

List: Inserção de elementos com o método `Add()`

- Resultado:



List: Inserção de elementos com o método `Insert()`

- O método `Insert()` insere um elemento num índice específico da *list*

- Sintaxe:

`Insert(índice, elemento)`

- Em que:

`índice` O índice numérico onde o elemento é inserido

`elemento` O elemento a inserir

- Exemplo:

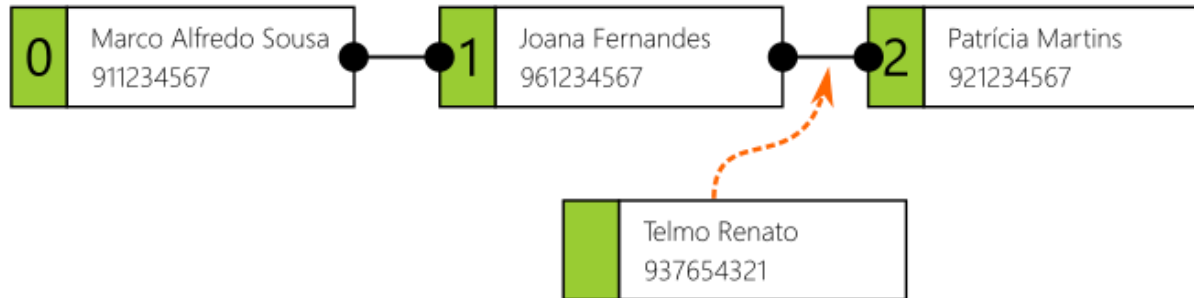
```
List<Cliente> clientes = new List<Cliente>();

// Adicionar clientes
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));
clientes.Add(new Cliente("Joana Fernandes", 961234567));
clientes.Add(new Cliente("Patrícia Martins", 921234567));

// Inserir um novo cliente entre os dois últimos clientes
clientes.Insert(2, new Cliente("Telmo Renato", 937654321));
```

List: Inserção de elementos com o método `Insert()`

Antes do método `Insert()`



Após o método `Insert()`



List: Remoção de elementos

- Para remover elementos de uma *list* podem ser utilizados vários métodos, entre os quais:
 - o método `RemoveAt()`: remove um elemento com um determinado índice
 - o método `RemoveRange()`: remove um conjunto de elementos

List: Remoção de elementos com o método `RemoveAt()`

- Este método remove um elemento com um determinado índice
- Sintaxe:

`RemoveAt(índice)`

- Em que:

`índice` O índice numérico do elemento a remover

- Exemplo:

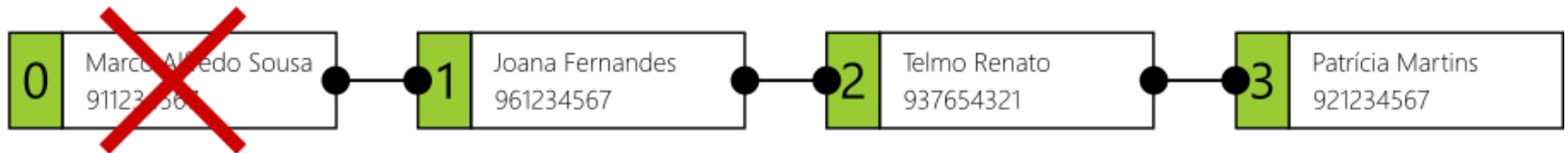
```
List<Cliente> clientes = new List<Cliente>();

// Adicionar clientes
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));
clientes.Add(new Cliente("Joana Fernandes", 961234567));
clientes.Add(new Cliente("Telmo Renato", 937654321));
clientes.Add(new Cliente("Patrícia Martins", 921234567));

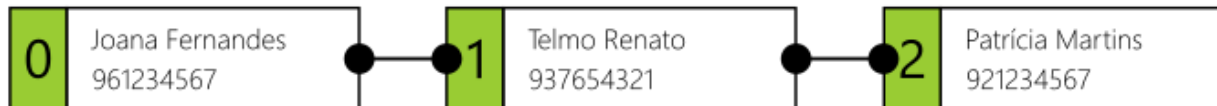
// Remover o primeiro elemento da lista de clientes
clientes.RemoveAt(0);
```

List: Remoção de elementos com o método `RemoveAt()`

Antes do método `RemoveAt()`



Após o método `RemoveAt()`



List: Remoção de elementos com o método `RemoveRange()`

- Este método remove um conjunto de elementos, a partir de um determinado índice

- Sintaxe:

`RemoveRange(índice, quantidade)`

- Em que:

`índice` O índice numérico a partir do qual são removidos os elementos

`quantidade` A quantidade de elementos a remover

- Exemplo:

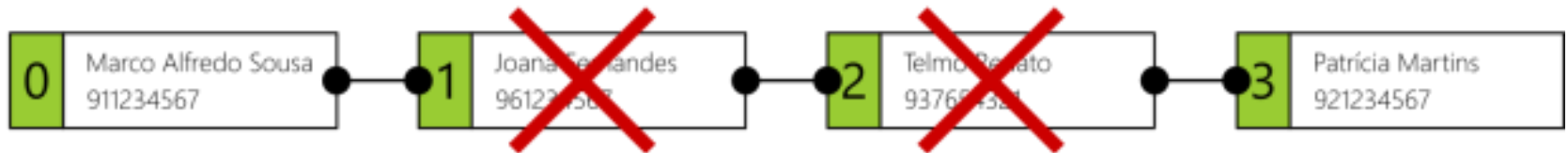
```
List<Cliente> clientes = new List<Cliente>();

// Adicionar clientes
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));
clientes.Add(new Cliente("Joana Fernandes", 961234567));
clientes.Add(new Cliente("Telmo Renato", 937654321));
clientes.Add(new Cliente("Patrícia Martins", 921234567));

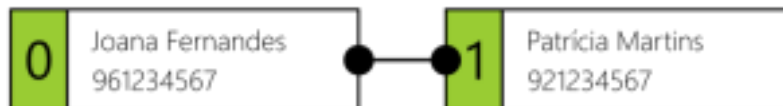
// Remover dois elementos da lista de clientes
clientes.RemoveRange(1, 2);
```

List: Remoção de elementos com o método `RemoveRange()`

Antes do método `RemoveRange()`



Após o método `RemoveRange()`



List: Aceder a um elemento

- Para aceder a um elemento de uma *list* é utilizada a seguinte sintaxe:

`nome[índice]`

- Em que:

`nome` Nome da *list*

`índice` Índice do elemento a aceder

- Exemplo 1: obter uma parte do valor (o peso do produto)

```
List<Cliente> clientes = new List<Cliente>();  
  
// Adicionar clientes  
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));  
clientes.Add(new Cliente("Joana Fernandes", 961234567));  
clientes.Add(new Cliente("Patrícia Martins", 921234567));  
  
// Obter o nome do primeiro cliente  
Console.WriteLine($"Nome cliente: {clientes[0].nome}");
```

List: Aceder a um elemento

- Exemplo 2: obter os dados completos do segundo cliente

```
List<Cliente> clientes = new List<Cliente>();  
  
// Adicionar clientes  
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));  
clientes.Add(new Cliente("Joana Fernandes", 961234567));  
clientes.Add(new Cliente("Patrícia Martins", 921234567));  
  
// Obter os dados completos do segundo cliente  
Cliente dadosCliente = clientes[1];  
Console.WriteLine($"Nome cliente: {dadosCliente.nome}");  
Console.WriteLine($"Telemóvel cliente: {dadosCliente.telemovel}");
```

List: Percorrer os elementos com `foreach`

- Uma forma de percorrer uma *list* é utilizando a instrução `foreach`:

```
List<Cliente> clientes = new List<Cliente>();

// Adicionar clientes
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));
clientes.Add(new Cliente("Joana Fernandes", 961234567));
clientes.Add(new Cliente("Patrícia Martins", 921234567));

// Percorrer a list
Console.WriteLine("Conteúdo da lista:");
foreach (Cliente c in clientes)
{
    Console.WriteLine($"Nome: {c.nome}    Telemóvel: {c.telemovel}");
}
```

List: Percorrer os elementos através de índice

- Uma forma de percorrer uma *list* é utilizando a instrução **for**:

```
List<Cliente> clientes = new List<Cliente>();

// Adicionar clientes
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));
clientes.Add(new Cliente("Joana Fernandes", 961234567));
clientes.Add(new Cliente("Patrícia Martins", 921234567));

// Percorrer a list
Console.WriteLine("Conteúdo da lista:");
for (int i = 0; i < clientes.Count; ++i)
{
    Console.WriteLine($"Nome: {clientes[i].nome} Telemóvel:
{clientes[i].telemovel}");
}
```

List: Obter o número de elementos

- A propriedade `Count` devolve o número de elementos numa *list*

```
List<Cliente> clientes = new List<Cliente>();  
  
// Adicionar clientes  
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));  
clientes.Add(new Cliente("Joana Fernandes", 961234567));  
clientes.Add(new Cliente("Patrícia Martins", 921234567));  
  
// Número de elementos na lista  
Console.WriteLine($"A lista possui {clientes.Count} elementos");
```

List: Verificar se contém um elemento

- O método `Contains()` é utilizado para verificar se uma *list* contém um determinado elemento:

`Contains(elemento)`

- Em que:

`elemento` O elemento a verificar se existe na *list*

- Este método retorna um valor booleano:

`true` Se o elemento foi encontrado

`false` Se o elemento não foi encontrado

List: Verificar se contém um elemento

- Exemplo:

```
List<Cliente> clientes = new List<Cliente>();

// Adicionar clientes
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));
clientes.Add(new Cliente("Joana Fernandes", 961234567));
clientes.Add(new Cliente("Patrícia Martins", 921234567));

// Verificar se a lista contém um determinado elemento
if (clientes.Contains(new Cliente("Patrícia Martins", 921234567)))
{
    Console.WriteLine("A lista contém o elemento");
}
else
{
    Console.WriteLine("A lista não contém o elemento");
}
```

List: Limpar o conteúdo

- Para remover o conteúdo de uma *list* é utilizado o método `Clear()`

```
List<Cliente> clientes = new List<Cliente>();

// Adicionar clientes
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));
clientes.Add(new Cliente("Joana Fernandes", 961234567));
clientes.Add(new Cliente("Patrícia Martins", 921234567));

// Número de elementos na lista
Console.WriteLine($"A lista possui {clientes.Count} elementos");

// Limpar a stack
clientes.Clear();

// Número de elementos na lista (0 elementos)
Console.WriteLine($"A lista possui {clientes.Count} elementos");
```

List: Inverter a ordem dos elementos

- O método `Reverse()` inverte a ordem dos elementos numa *list*
- Sintaxe:

`Reverse()`

Antes do método `Reverse()`



Após o método `Reverse()`



List: Inverter a ordem dos elementos

- Exemplo:

```
List<Cliente> clientes = new List<Cliente>();

// Adicionar clientes
clientes.Add(new Cliente("Marco Alfredo Sousa", 911234567));
clientes.Add(new Cliente("Joana Fernandes", 961234567));
clientes.Add(new Cliente("Patrícia Martins", 921234567));

// Inverter a ordem dos elementos da lista
clientes.Reverse();

Console.WriteLine("Conteúdo da lista após Reverse()");
foreach (Cliente c in clientes)
{
    Console.WriteLine($"Nome: {c.nome}    Telemóvel: {c.telemovel}");
}
```

| *Dictionary*

Dictionary

- Um *dictionary* (dicionário) é uma estrutura composta por duas partes:
 - uma **chave** (*key*)
 - um **valor** (*value*)
- A chave é utilizada para ter acesso ao valor
- É uma estrutura muito utilizada porque frequentemente os dados são identificados e acedidos através de uma chave (como um código)
- Por exemplo:
 - dados de um produto, em que o produto é identificado através de um código
 - dados de um cliente, em que o cliente é identificado através de um código
 - dados de um automóvel, em que este é identificado através da matrícula
 - etc.

Dictionary

- Imaginemos um conjunto de produtos:
 - cada produto é identificado por um código (a chave)
 - cada produto contém dados (o valor)

Chave
(código produto)

Valor
(dados produto)

82165202



Designação: Ar condicionado fixo Hair
Nebula Green BL2000F

Quantidade: 8

Peso: 27 Kg

B071GYHSFX



Designação: Balança digital Active Era Ultra Slim

Quantidade: 11

Peso: 1.2 Kg

Dictionary: Declaração

- Para criar um novo *dictionary* utiliza-se a seguinte sintaxe:

```
Dictionary<tChave, tValor> nome = new Dictionary<tChave, tValor>()
```

- Em que:

tChave O tipo de dados da **chave** (*int*, *string*, etc.)

tValor O tipo de dados do **valor** (*int*, *string*, etc.)

nome O nome do *dictionary*

Dictionary: Declaração

- No seguinte exemplo, é criado um dicionário para armazenar dados de produtos em que:
 - a chave é do tipo `string`
 - o valor é uma `struct` do tipo `Produto`

```
public struct Produto
{
    public string designacao;
    public int quantidade;
    public double peso;

    public Produto(string _designacao, int _quantidade, double _peso)
    {
        designacao = _designacao;
        quantidade = _quantidade;
        peso = _peso;
    }
}
```

Tipo da chave

Tipo do valor

Nome do *dictionary*

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();
```

Dictionary: Inserção de elementos

- Para inserir elementos num *dictionary* é utilizado o método `Add()`

- Sintaxe:

`Add(chave, valor)`

- Em que:

`chave` A chave do elemento a inserir

`valor` O valor do elemento a inserir

- Exemplo:

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();  
  
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green  
BL2000F", 8, 27.0D));  
  
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra  
Slim", 11, 1.2D));
```

Dictionary: Inserção de elementos

- Para inserir elementos num *dictionary* é utilizado o método `Add()`

- Sintaxe:

`Add(chave, valor)`

- Em que:

`chave` A chave do elemento a inserir

`valor` O valor do elemento a inserir

- Exemplo:

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();  
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green  
BL2000F", 8, 27.0D));  
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra  
Slim", 11, 1.2D));
```

Chave

Chave

Dictionary: Inserção de elementos

- Para inserir elementos num *dictionary* é utilizado o método `Add()`

- Sintaxe:

`Add(chave, valor)`

- Em que:

`chave` A chave do elemento a inserir

`valor` O valor do elemento a inserir

- Exemplo:

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();  
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green  
BL2000F", 8, 27.0D));  
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra  
Slim", 11, 1.2D));
```

Valor

Valor

Dictionary

Exercício

- Crie um *dictionary* chamado `dicionario`
- O objetivo desta estrutura será funcionar como um verdadeiro dicionário
- Para tal:
 - a chave deverá ser do tipo `string`
 - o valor deverá ser do tipo `string`
- Insira os seguintes dados (chave:valor) na estrutura de dados que criou:
 - "epifania": "Apreensão, geralmente inesperada, do significado de algo"
 - "desfasamento": "Ausência de sintonia"
 - "cinéfilo": "Que ou quem tem forte interesse ou entusiasmo pelo cinema"

Dictionary

Exercício: Resolução

```
Dictionary<string, string> dicionario = new Dictionary<string, string>();  
  
dicionario.Add("epifania", "Apreensão, geralmente inesperada, do significado de  
algo");  
dicionario.Add("desfasamento", "Ausência de sintonia");  
dicionario.Add("cinéfilo", "Que ou quem tem forte interesse ou entusiasmo pelo  
cinema");
```

Dictionary: O método `ContainsKey()`

- Ao tentar inserir um elemento com uma chave já existente, é provocado um erro:

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();  
  
// Adicionar produtos  
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green  
BL2000F", 8, 27.0D));  
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra  
Slim", 11, 1.2D));  
  
// Tentar inserir um novo elemento com chave já existente  
// (o seguinte código irá provocar um erro)  
produtos.Add("82165202", new Produto("Automóvel Toyota", 20, 1200.0D));
```

- A solução passa por verificar utilizando o método `ContainsKey()` se a chave já existe, antes de inserir o novo elemento:

```
if (!produtos.ContainsKey("82165202"))  
{  
    produtos.Add("82165202", new Produto("Automóvel Toyota", 20, 1200.0D));  
}
```

Dictionary: Remoção de elementos

- Para remover elementos de um *dictionary* é utilizado o método `Remove()`
- Sintaxe:
`Remove(chave)`
- Em que:
`chave` A chave do elemento a remover
- Este método retorna um valor booleano:
`true` Se o elemento com a chave especificada foi encontrado e removido com sucesso
`false` Se o elemento não foi encontrado

Dictionary: Remoção de elementos

- Exemplo:

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();

// Adicionar produtos
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green
BL2000F", 8, 27.0D));
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra
Slim", 11, 1.2D));

// Remover o produto com o código "82165202"
bool resultado = produtos.Remove("82165202");

if (resultado)
{
    Console.WriteLine("O produto com código 82165202 foi removido com
sucesso.");
}
else
{
    Console.WriteLine("O produto com código 82165202 não foi removido.");
}
```

Dictionary: Aceder a um elemento

- Para aceder a um elemento de um *dictionary* é utilizada a seguinte sintaxe:

`nome[chave]`

- Em que:

`nome` Nome do *dictionary*

- Exemplo 1: obter uma parte do valor (o peso do produto)

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();

// Adicionar produtos
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green
BL2000F", 8, 27.0D));
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra
Slim", 11, 1.2D));

// Obter o peso do elemento com código "B071GYHSFX" (1.2 Kg)
double peso = produtos["B071GYHSFX"].peso;
Console.WriteLine($"Peso do produto: {peso} Kg");
```

Dictionary: Aceder a um elemento

- Exemplo 2: obter o valor na totalidade (todos os dados do produto)

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();  
  
// Adicionar produtos  
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green  
BL2000F", 8, 27.0D));  
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra  
Slim", 11, 1.2D));  
  
// Obter o elemento com código "B071GYHSFX"  
Produto p1 = produtos["B071GYHSFX"];  
Console.WriteLine($"Designação do produto: {p1.designacao}");  
Console.WriteLine($"Quantidade disponível: {p1.quantidade} unidade");  
Console.WriteLine($"Peso do produto: {p1.peso} Kg");
```

Dictionary: Aceder a um elemento

- Se tentarmos aceder a um elemento cuja chave não existe, ocorre um erro:

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();  
  
// Adicionar produtos  
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green  
BL2000F", 8, 27.0D));  
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra  
Slim", 11, 1.2D));  
  
// Tentar mostrar a quantidade do elemento com código "abc123"  
Console.WriteLine(produtos["abc123"].quantidade);
```

Não existe um elemento
com a chave "abc123"

Dictionary: Aceder a um elemento

- Neste caso, devemos utilizar o método `TryGetValue()` para tentar aceder ao elemento com a chave especificada:
 - se a chave existir, o valor do elemento é atribuído a um *out parameter*
 - se a chave não existir, não ocorre erro

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();  
  
// Adicionar produtos  
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green  
BL2000F", 8, 27.0D));  
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra  
Slim", 11, 1.2D));  
  
// Tentar mostrar a quantidade do elemento com código "abc123"  
// (não irá provocar erro, pois é utilizado o método TryGetValue())  
Produto p2;  
if (produtos.TryGetValue("abc123", out p2))  
{  
    Console.WriteLine(p2.quantidade);  
}
```

Apenas é executado se
existir a chave "abc123"

Dictionary: Modificar um elemento

- Para modificar os dados de um elemento, por exemplo, modificar o peso de um produto:

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();

// Adicionar produtos
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green
BL2000F", 8, 27.0D));
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra
Slim", 11, 1.2D));

// Obter o elemento com código "B071GYHSFX"
Produto p1 = produtos["B071GYHSFX"];

// Modificar o peso do produto
p1.peso = 1.4D;

// Substituir o valor do elemento com código "B071GYHSFX"
produtos["B071GYHSFX"] = p1;
```

Dictionary: Percorrer os elementos

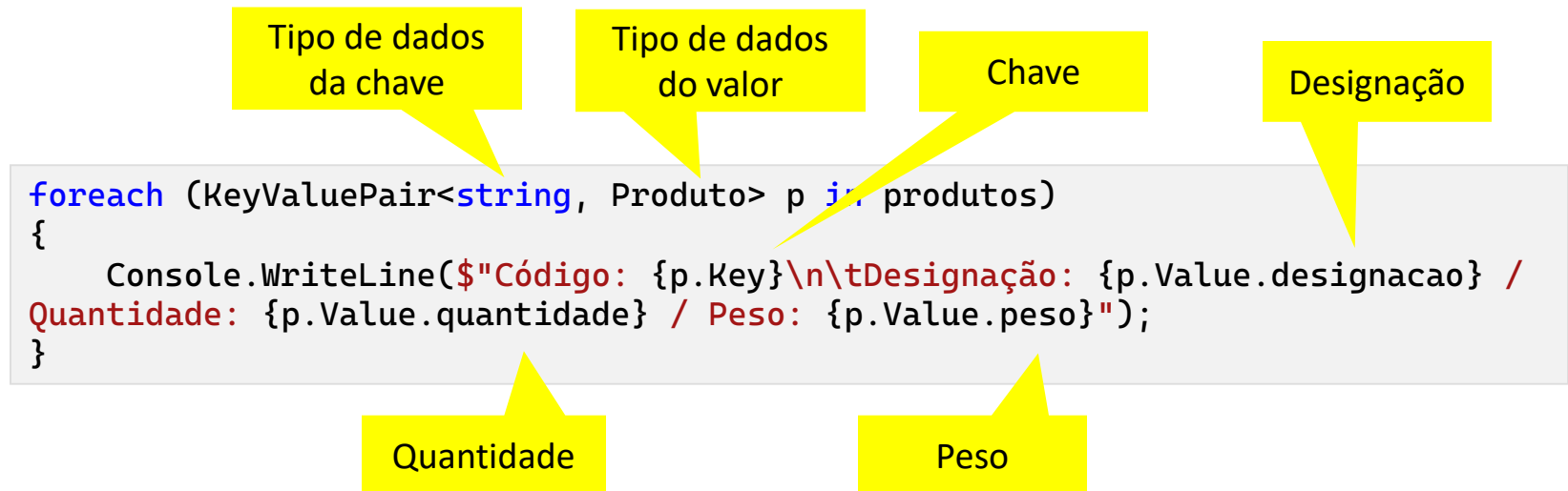
- A forma de percorrer um *dictionary* pode inicialmente parecer complexo:

```
foreach (KeyValuePair<string, Produto> p in produtos)
{
    Console.WriteLine($"Código: {p.Key}\n\tDesignação: {p.Value.designacao} /
    Quantidade: {p.Value.quantidade} / Peso: {p.Value.peso}");
}
```

- No entanto, após análise mais atenta verificamos que esta estrutura permite guardar dados com alguma complexidade e esta solução é uma forma elegante de ter acesso a todos os dados

Dictionary: Percorrer os elementos

- A forma de percorrer um *dictionary* pode inicialmente parecer complexo:



The diagram illustrates the components of a C# dictionary iteration. A code block is shown with yellow callout boxes pointing to specific parts of the code:

- Tipo de dados da chave** points to `string` in `KeyValuePair<string, Produto>`.
- Tipo de dados do valor** points to `Produto` in `KeyValuePair<string, Produto>`.
- Chave** points to `p.Key` in the `Console.WriteLine` statement.
- Designação** points to `p.Value.designacao` in the `Console.WriteLine` statement.
- Quantidade** points to `p.Value.quantidade` in the `Console.WriteLine` statement.
- Peso** points to `p.Value.peso` in the `Console.WriteLine` statement.

```
foreach (KeyValuePair<string, Produto> p in produtos)
{
    Console.WriteLine($"Código: {p.Key}\n\tDesignação: {p.Value.designacao} /
    Quantidade: {p.Value.quantidade} / Peso: {p.Value.peso}");
}
```

- No entanto, após análise mais atenta verificamos que esta estrutura permite guardar dados com alguma complexidade e esta solução é uma forma elegante de ter acesso a todos os dados

Dictionary: Obter o número de elementos

- O método `Count()` devolve o número de elementos num *dictionary*:

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();  
  
// Adicionar produtos  
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green  
BL2000F", 8, 27.0D));  
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra  
Slim", 11, 1.2D));  
  
// Número de elementos no dictionary  
Console.WriteLine($"O dictionary possui {produtos.Count()} elementos");
```

Dictionary: Limpar o conteúdo

- Para limpar o conteúdo de um *dictionary* é utilizado o método `Clear()`: este método remove todos os elementos

```
Dictionary<string, Produto> produtos = new Dictionary<string, Produto>();

// Adicionar produtos
produtos.Add("82165202", new Produto("Ar condicionado fixo Hair Nebula Green
BL2000F", 8, 27.0D));
produtos.Add("B071GYHSFX", new Produto("Balança digital Active Era Ultra
Slim", 11, 1.2D));

// Número de elementos no dictionary
Console.WriteLine($"O dictionary possui {produtos.Count()} elementos");

// Limpar o dictionary
produtos.Clear();

// Número de elementos no dictionary (0 elementos)
Console.WriteLine($"O dictionary possui {produtos.Count()} elementos");
```