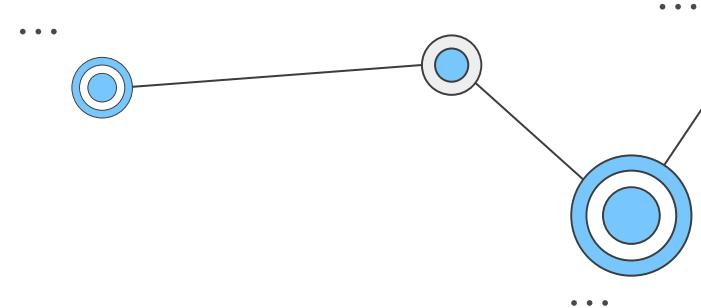


# Entendendo Tecnologias Microsoft

A ideia é mostrar o funcionamento  
por debaixo do capô das  
tecnologias Microsoft



...

...

...

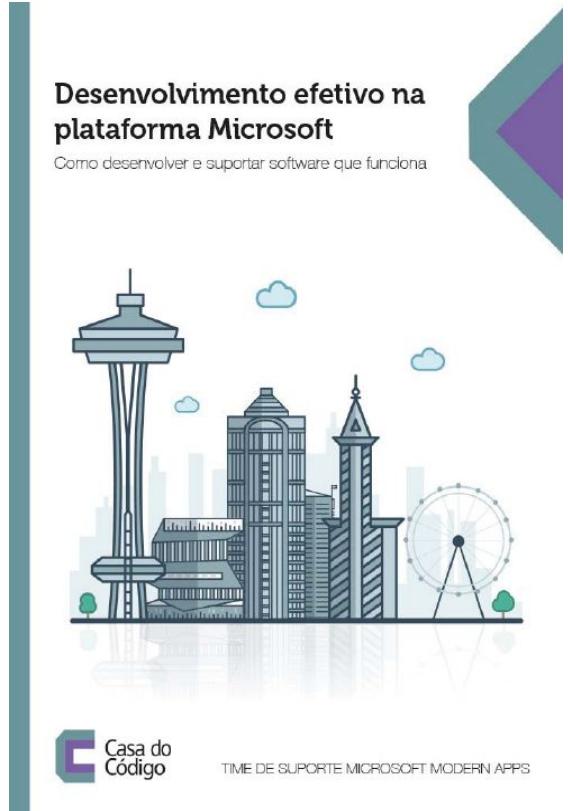
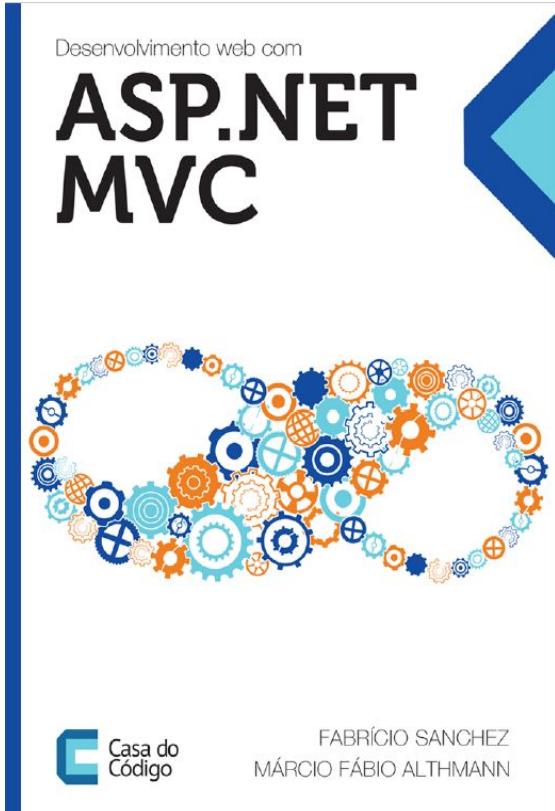
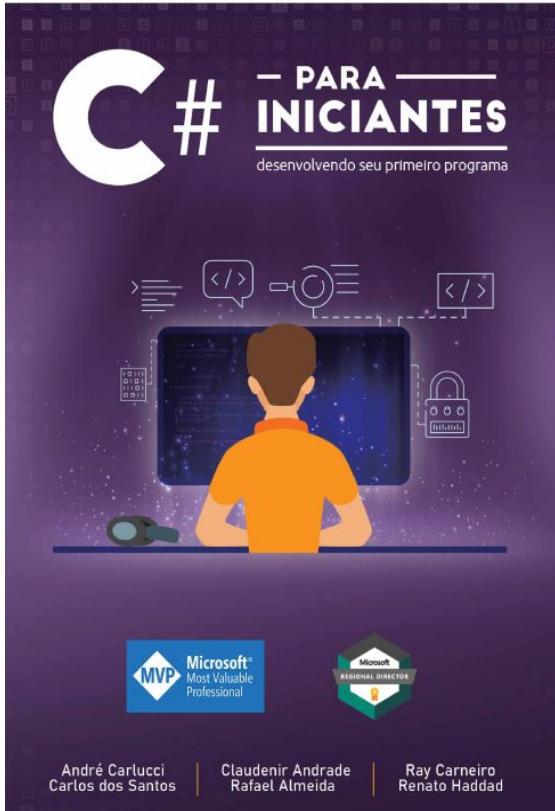
# Olá!

## Eu Sou Willian Brito!



- ❖ Desenvolvedor FullStack na Msystem Software
  - ❖ Formado em Analise e Desenvolvimento de Sistemas.
  - ❖ Pós Graduado em Segurança Cibernética.
  - ❖ Certificação SYCP (Solyd Certified Pentester) v2018
- ...
- ...
- ...

# Este conteúdo é baseado nessas obras





“Este **conteúdo** é para pessoas que querem **entender o funcionamento** das **tecnologias** envolvidas no **ambiente da microsoft**”



## Sistema Operacional

01

Windows é uma família de sistemas operacionais desenvolvidos pela Microsoft.

...

## Linguagem de Programação

02

C# é uma linguagem de programação, desenvolvida pela Microsoft como parte da plataforma .NET.

...

## .NET Framework

03

O .NET Framework é uma iniciativa da empresa Microsoft, que visa uma plataforma única para desenvolvimento e execução de aplicações.

...

## Servidor Web

04

IIS é um servidor web criado pela Microsoft para seus sistemas operacionais para servidores.

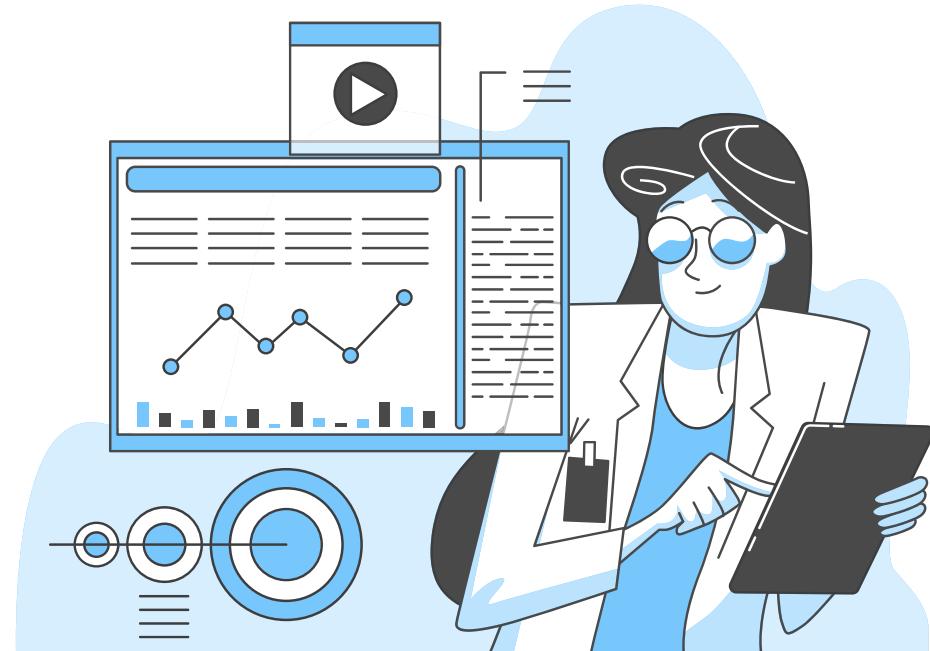
...

## Banco de Dados

05

O Microsoft SQL Server é um sistema gerenciador de Banco de dados relacional desenvolvido pela Sybase em parceria com a Microsoft.

# Resumo das Tecnologias

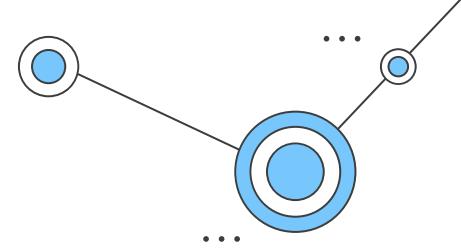


# 01

## Sistema Operacional

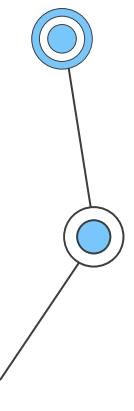
A base sobre a qual os softwares e serviços são escritos e executados.

# Microsoft Windows



**Windows** é uma família de **sistemas operacionais** desenvolvidos, comercializados e vendidos pela **Microsoft**. É constituída por várias famílias de sistemas operacionais, sob o comando de Bill Gates e Paul Allen, o Windows já passou por diversas atualizações, partindo do número 1 até o 11, sendo este último o mais recente criado pela empresa.

Seu objetivo era facilitar o uso do computador graças a um conceito e ferramenta revolucionários: a interface gráfica e o mouse. Em vez de memorizar e digitar comandos complexos, bastava apontar para opções na tela.



Não foi um sucesso imediato, mas ao longo de vários anos e múltiplas versões o sistema foi crescendo e conquistando espaço, até dominar completamente a computação pessoal. Não faltam alternativas, como o Linux e mac OS, mas para muitas pessoas o Windows é sinônimo de "PC", e é inconcebível que um possa existir sem o outro.



# Versões do Windows

Dentro dos 40 anos de história do software, o Windows já disponibilizou diversas versões do sistema operacional, que as vezes agradaram ao público consumidor e outras nem tanto. Em meio aos sucessos e as opções menos bem recebidas aqui vamos listar as principais versões do Windows:

1981  
985

Windows 1.0

Lançaram Interface Gráfica.

1990  
1990

Windows 3.1

Conexão de Rede Local.

1993

Windows NT (32 bits)

Windows Totalmente Novo Construirão um Kernel do 0 e pela primeira vez implementaram em um Windows a arquitetura 32 bits.

1995

Windows 95

Trouxe uma novidade incrível o Menu Iniciar Interface gráfica nova.



# Versões do Windows

1998

Windows 98

Evolução do Windows 95.

2000

Windows ME / Windows 2000

Surgiram 2 Windows o Millennium que é a evolução do Windows 98 e o 2000 que é a evolução da Família NT.

2001

Windows XP (64 bits):  
Fizeram uma Mudança Drástica Encerraram a linha da família do Windows 1, 3, 95, 98, ME e continuaram da família 2000 que é mais voltada para servidores corporativo e agora todos usuários usavam Windows tanto para usuários caseiros quanto para empresas corporativa.

Esse Windows foi uma Grande Revolução que é lembrado até hoje, eles Implementaram a tecnologia de Multusuários e Permissões a esses Usuários. Lembrando que esse foi o primeiro Windows com Arquitetura de 64 bits e Sistemas de Arquivos NTFS.



# Versões do Windows

2003

**Windows Server 2003**  
Constitui-se, no seu núcleo, de uma versão do Windows XP com algumas simplificações realizadas com o objetivo de fornecer um funcionamento mais estável do sistema e maior segurança para trabalhos em redes.

2006

**Windows Vista**

Essa versão apresenta uma nova interface gráfica do usuário, assim como novas funções para busca e manipulação de artefatos para aplicações multimídia e para redes de comunicação, esta versão não foi bem aceita pelo público.

2008

**Windows Server 2008**

Provê uma plataforma para construção de uma infraestrutura para desenvolvimento de aplicações envolvendo ambientes de redes e web services.

2009

**Windows 7**

Tinha a intenção de continuar compatível com aplicações e hardwares nos quais o Windows Vista já funcionava, mas com correções e melhor desempenho.



# Versões do Windows

2012

## Windows 8

Sistema para qualquer dispositivo, comum a interface totalmente nova, adaptada para dispositivos sensíveis ao toque, houve algumas críticas por parte do público pela mudança muito radical na interface gráfica.

2015

## Windows 10

Finalmente chegamos ao Windows 10. O principal destaque desta versão foi a reversão da interface para o tradicional paradigma desktop do Windows 7, com Barra de Tarefas e Menu Iniciar. Recursos do Windows 8 como os apps modernos e a loja de aplicativos persistem, mas de forma muito mais integrada à interface tradicional, e muito mais familiar aos usuários de longa data.

2021

## Windows 11

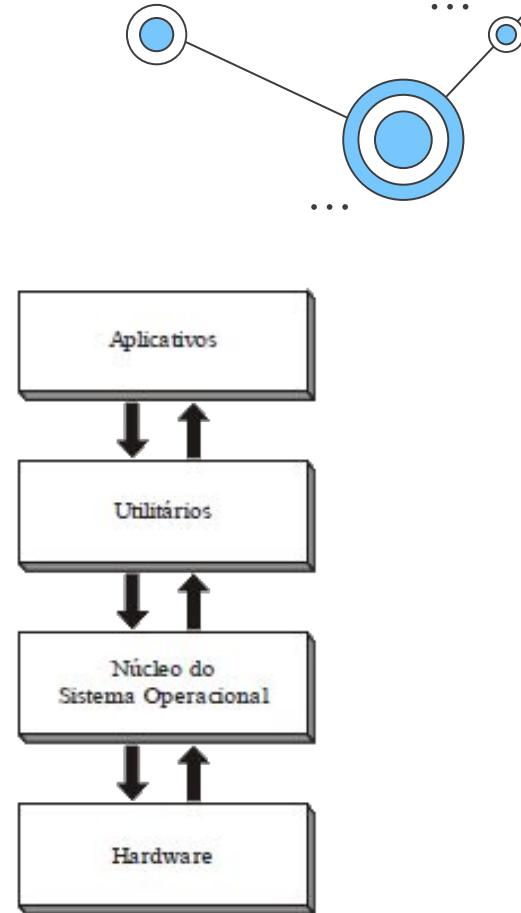
O Windows 11 é a mais nova versão e apresenta uma interface de usuário atualizada que segue as diretrizes do Fluent Design da Microsoft; translucidez, sombras e cantos arredondados prevalecem em todo o sistema. Um menu Iniciar redesenhado é usado, o que elimina os ladrilhos do lado direito. A barra de tarefas também é simplificada e centralizada por padrão.

# Estrutura de um S.O

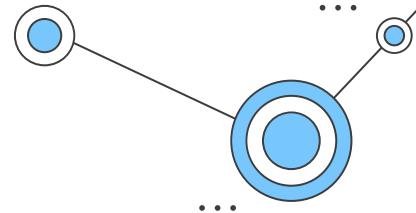
## □ Introdução

O sistema operacional é formado por um conjunto de rotinas que oferecem serviços aos usuários, às suas aplicações, e também ao próprio sistema. Esse conjunto de rotinas é denominado **núcleo do sistema** ou **kernel**.

É importante não confundir o núcleo do sistema com aplicações utilitários ou o interpretador de comandos, que acompanham o sistema operacional. As aplicações são utilizadas pelos usuários e escondem todos os detalhes da interação com o sistema. Os utilitários, como compiladores e editores de texto, e interpretadores de comandos permitem aos usuários, administradores e desenvolvedores uma interação amigável com o sistema.



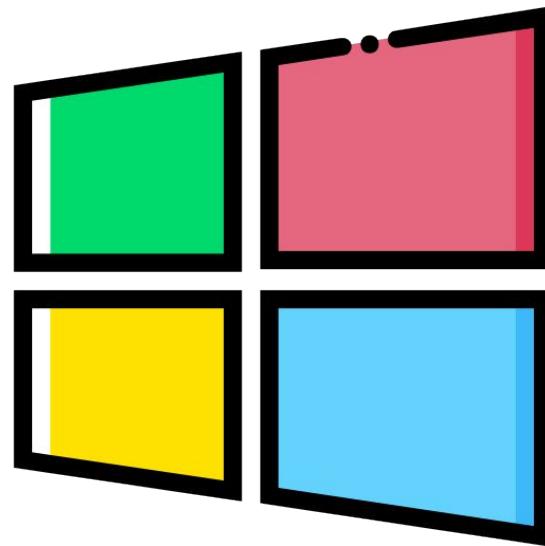
# Estrutura de um S.O



Existe uma grande dificuldade em compreender a estrutura e o funcionamento de um sistema operacional, pois ele não é executado como uma aplicação tipicamente sequencial, com inicio, meio e fim. Os procedimentos do sistema são executados concorrentemente sem uma ordem, com base em eventos dissociados do tempo (eventos assíncronos). Muitos desses eventos estão relacionados ao hardware e a tarefas internas do próprio sistema operacional.

## Principais Funções:

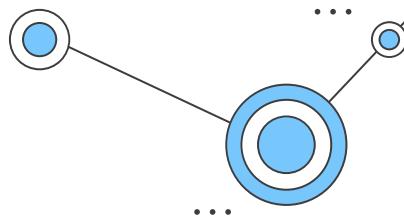
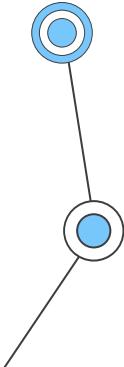
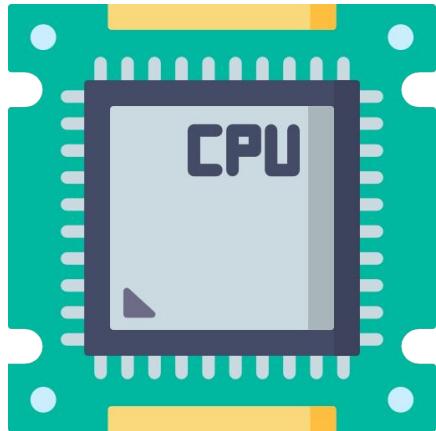
- Tratamento de interrupções e exceções;
- Criação e eliminação de processos e threads;
- Sincronização e comunicação entre processos e threads;
- Escalonamento e controle dos processos e threads;
- Gerencia de memória;
- Gerencia do sistema de arquivos;
- Gerencia de dispositivos de E/S;
- Suporte a redes locais e distribuídas;
- Contabilização do uso do sistema;
- Auditoria e segurança do sistema.



# Funcionalidades

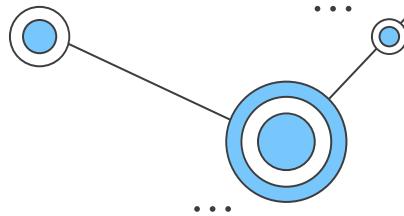
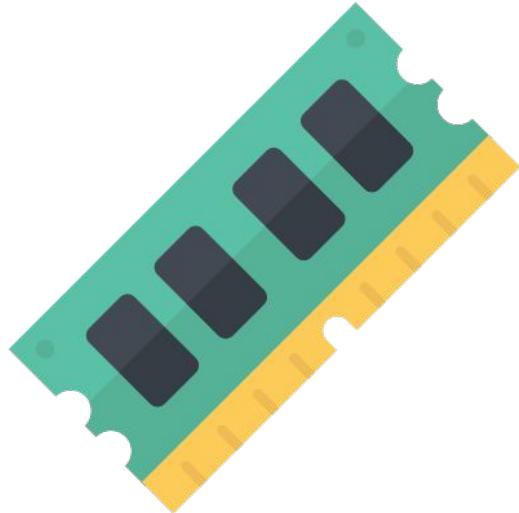
Para cumprir seus objetivos de abstração e gerência, o sistema operacional deve atuar em várias frentes. Cada um dos recursos do sistema possui suas particularidades, o que impõe exigências específicas para gerenciar e abstrair os mesmos. Sob esta perspectiva, as principais funcionalidades implementadas por um sistema operacional típico são:

**Gerenciamento do processador:** esta funcionalidade, também conhecida como gerência de processos, de tarefas ou de atividades, visa distribuir a capacidade de processamento de forma justa entre as aplicações, evitando que uma aplicação monopolize esse recurso e respeitando as prioridades definidas pelos usuários. O sistema operacional provê a ilusão de que existe um processador independente para cada tarefa, o que facilita o trabalho dos programadores de aplicações e permite a construção de sistemas mais interativos. Também faz parte da gerência de atividades fornecer abstrações para sincronizar atividades interdependentes e prover formas de comunicação entre elas.



# Funcionalidades

**Gerenciamento de memória:** tem como objetivo fornecer a cada aplicação uma área de memória própria, independente e isolada das demais aplicações e inclusive do sistema operacional. O isolamento das áreas de memória das aplicações melhora a estabilidade e segurança do sistema como um todo, pois impede aplicações com erros (ou aplicações maliciosas) de interferir no funcionamento das demais aplicações. Além disso, caso a memória RAM existente seja insuficiente para as aplicações, o sistema operacional pode aumentá-la de forma transparente às aplicações, usando o espaço disponível em um meio de armazenamento secundário (como um disco rígido). Uma importante abstração construída pela gerência de memória, com o auxílio do hardware, é a noção de memória virtual, que desvincula os endereços de memória vistos por cada aplicação dos endereços acessados pelo processador na memória RAM. Com isso, uma aplicação pode ser carregada em qualquer posição livre da memória, sem que seu programador tenha de se preocupar com os endereços de memória onde ela irá executar.



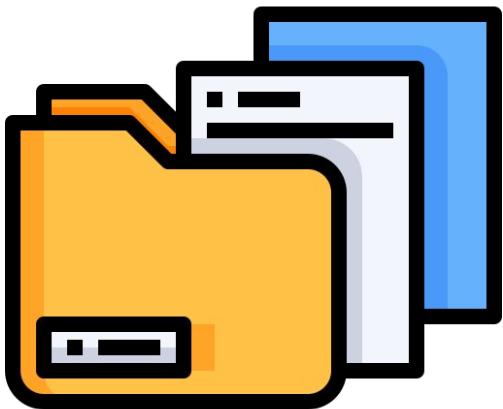
# Funcionalidades

**Gerenciamento de dispositivos:** cada periférico do computador possui suas particularidades assim, o procedimento de interação com uma placa de rede é completamente diferente da interação com um disco rígido SATA. Todavia, existem muitos problemas e abordagens em comum para o acesso aos periféricos. Por exemplo, é possível criar uma abstração única para a maioria dos dispositivos de armazenamento como pendrives, discos SATA ou IDE, CDROMs, etc., na forma de um vetor de blocos de dados. A função da gerência de dispositivos (também conhecida como gerência de entrada/saída) é implementar a interação com cada dispositivo por meio de drivers e criar modelos abstratos que permitam agrupar vários dispositivos similares sob a mesma interface de acesso.



# Funcionalidades

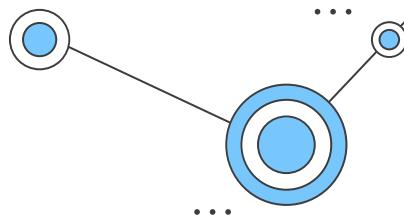
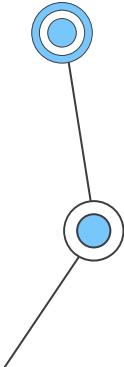
**Gerenciamento de arquivos:** esta funcionalidade é construída sobre a gerência de dispositivos e visa criar arquivos e diretórios, definindo sua interface de acesso e as regras para seu uso. É importante observar que os conceitos abstratos de arquivo e diretório são tão importantes e difundidos que muitos sistemas operacionais os usam para permitir o acesso a recursos que nada tem a ver com armazenamento. Exemplos disso são as conexões de rede (nos sistemas UNIX e Windows, cada socket TCP é visto como um descriptor de arquivo no qual pode-se ler ou escrever dados) e as informações internas do sistema operacional (como o diretório /proc do UNIX). No sistema experimental Plan 9, por exemplo, todos os recursos do sistema operacional são vistos como arquivos.



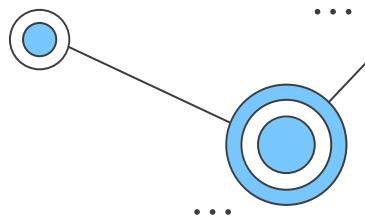
# Funcionalidades

**Gerenciamento de proteção:** com computadores conectados em rede e compartilhados por vários usuários, é importante definir claramente os recursos que cada usuário pode acessar, as formas de acesso permitidas (leitura, escrita, etc.) e garantir que essas definições sejam cumpridas. Para proteger os recursos do sistema contra acessos indevidos, é necessário:

- Definir usuários e grupos de usuários;
- Identificar os usuários que se conectam ao sistema, através de procedimentos de autenticação;
- Definir e aplicar regras de controle de acesso aos recursos, relacionando todos os usuários, recursos e formas de acesso e aplicando essas regras através de procedimentos de autorização
- Registrar o uso dos recursos pelos usuários, para fins de auditoria e contabilização.



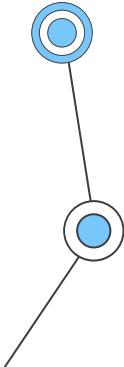
# System Calls



Uma preocupação que surge nos projetos de sistemas operacionais é a implementação de mecanismos de proteção ao núcleo do sistema e de acesso aos seus serviços. Caso uma aplicação que tenha acesso ao núcleo realize uma operação que altere sua integridade, todo o sistema poderá ficar comprometido.

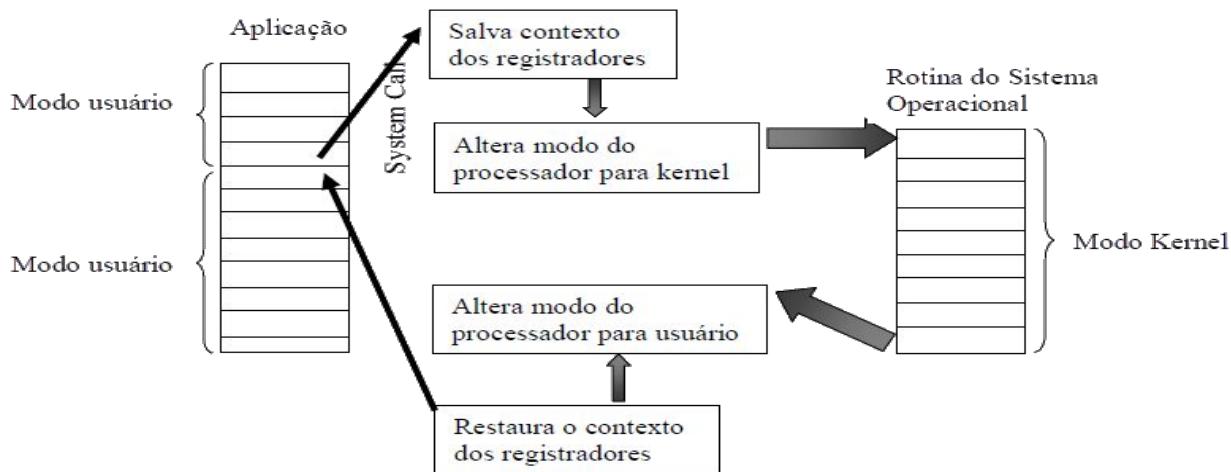
Todo o controle de execução de rotinas do SO é realizado pelo mecanismo conhecido como **system call**. Toda vez que uma aplicação desejar chamar uma rotina do SO, o mecanismo de system call é ativado. Inicialmente, o SO verificará se a aplicação possui privilégios necessários para executar a rotina desejada. Em caso negativo, o SO impedirá o desvio para a rotina do sistema, sinalizando ao programa chamador que a operação não é possível.

Considerando que a aplicação possua o devido privilégio para chamar a rotina do sistema desejada, o SO primeiramente salva o conteúdo corrente dos registradores, troca o modo de acesso do processador de usuário para kernel e realiza o desvio para a rotina alterando o registrador PC com o endereço da rotina chamada. Ao término da execução da rotina do sistema, o modo de acesso é alterado de kernel para usuário e o contexto dos registradores restaurados para que a aplicação continue a execução a partir da instrução que chamou a rotina do sistema.



# System Calls

Uma aplicação sempre deve executar com o processador em modo usuário. Caso uma aplicação tente executar diretamente uma instrução privilegiada sem ser por intermédio de uma chamada à rotina do sistema, um **mecanismo de proteção por hardware** garantirá a segurança do sistema, impedindo a operação. Nesta situação, em que a aplicação tenta executar uma instrução privilegiada em modo usuário e sem a supervisão do SO, o próprio hardware do processador sinalizará um erro. Uma exceção é gerada e a execução do programa é interrompida, protegendo desta forma o núcleo do sistema.



# System Calls

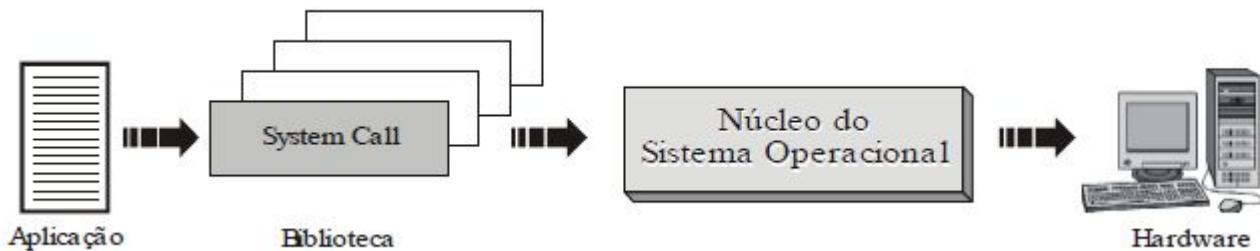
Os mecanismos de system call e de proteção por hardware garantem a segurança e a integridade do sistema. Com isso, as aplicações estão impedidas de executarem instruções privilegiadas sem a autorização e a supervisão do sistema operacional.

As system calls podem ser entendidas como uma porta de entrada para o acesso ao núcleo do sistema operacional e a seus serviços. Sempre que um usuário ou aplicação desejar algum serviço do sistema, é realizada uma chamada a uma de suas rotinas através de uma system call (chamada ao sistema). O termo **system call** é tipicamente utilizado em sistemas Unix, porém em outros sistemas o mesmo conceito é apresentado com diferentes nomes, como **system services** no Open VMS e **Application Program Interface (API)** no Windows da Microsoft.

# System Calls

Para cada serviço disponível existe uma system call associada e cada sistema operacional tem seu próprio conjunto de chamadas, com nomes, parâmetros e formas de ativação específicos. Isto explica por que uma aplicação desenvolvida utilizando serviços de um determinado sistema operacional não pode ser portada diretamente.

Através dos parâmetros fornecidos na system call, a solicitação é processada e uma resposta é retornada à aplicação juntamente com um estado de conclusão indicando se houve algum erro. O mecanismo de ativação e comunicação entre o programa e o sistema operacional é semelhante ao mecanismo implementado quando um programa chama uma sub-rotina. As system calls podem ser divididas por grupos de função.

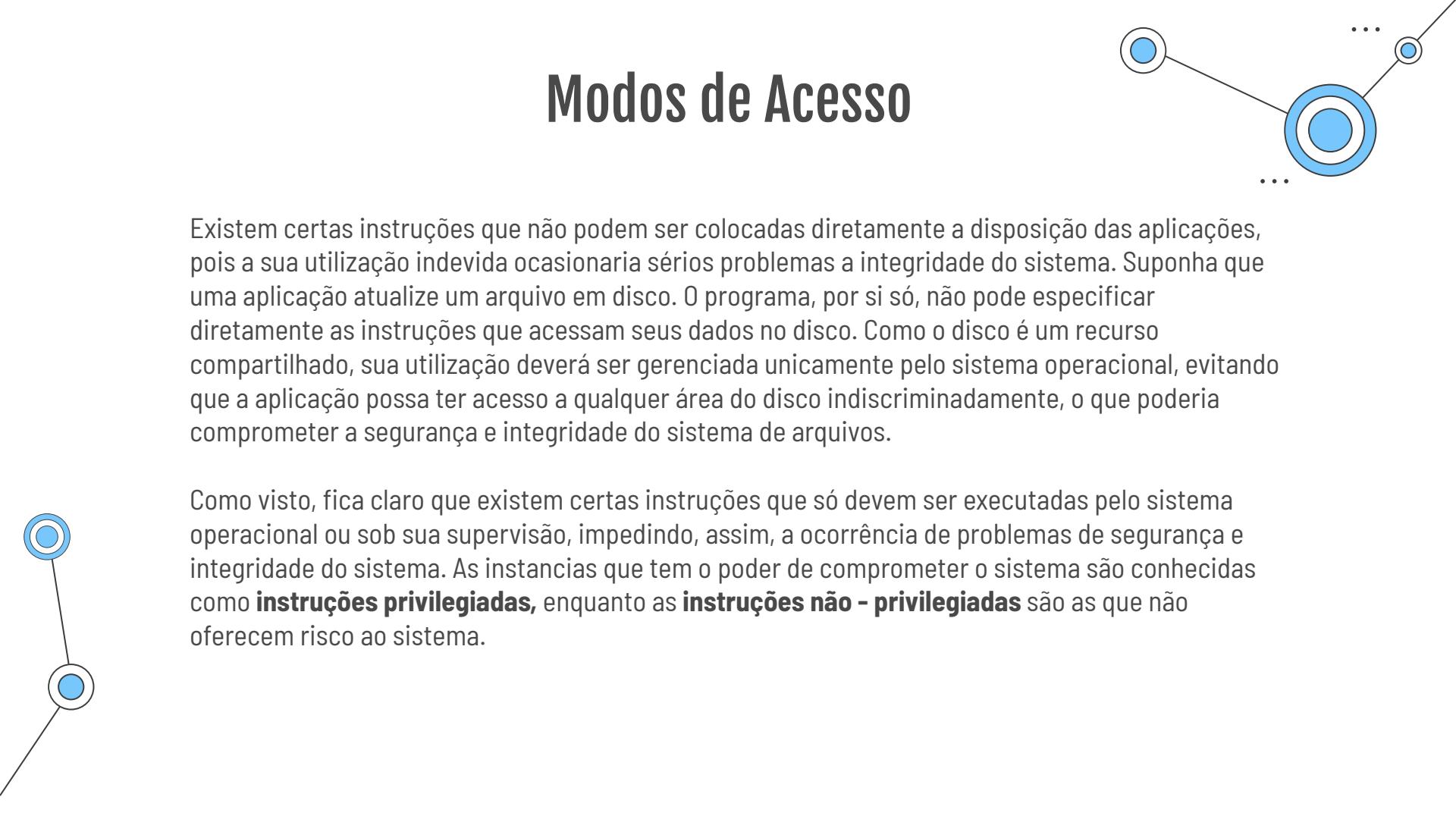


# System Calls

A maioria dos programadores e usuários desconhece os detalhes envolvidos, por exemplo, em um simples comando de leitura a um arquivo utilizando uma linguagem de alto nível. De forma simplificada, o comando da linguagem de alto nível é convertido pelo compilador para uma chamada a uma system call específica, que, quando executada, verifica a ocorrência de erros e retorna os dados ao programa de forma transparente ao usuário.

Funções	System Calls
Gerência de processo e threads	Criação e eliminação de processos e threads Alteração das características de processos e threads Sincronização e comunicação entre processos e threads Obtenção de informações sobre processos e threads
Gerência de memória	Alocação e desalocação de memória
Gerência do sistema de arquivos	Criação e eliminação de arquivos e diretórios Alteração das características de arquivos e diretórios Abrir e fechar arquivos Leitura e gravação em arquivos Obtenção de informações sobre arquivos e diretórios
Gerência de dispositivos	Alocação e desalocação de dispositivos Operações de entrada/saída em dispositivos Obtenção de informações sobre dispositivos

# Modos de Acesso

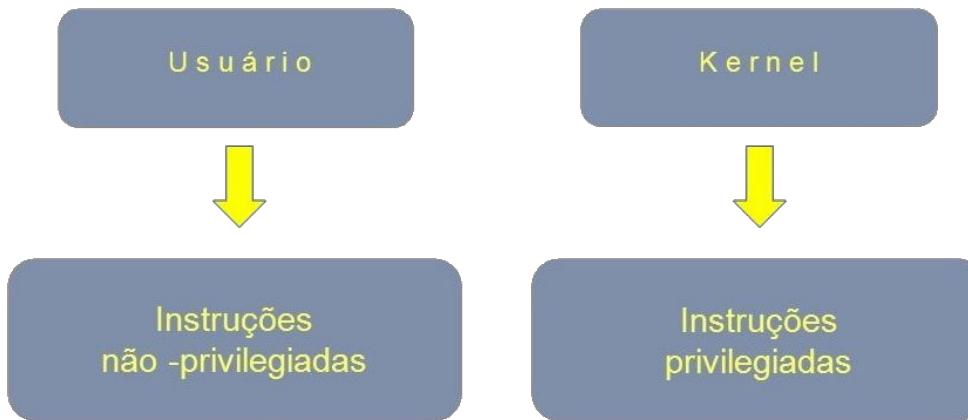


Existem certas instruções que não podem ser colocadas diretamente a disposição das aplicações, pois a sua utilização indevida ocasionaria sérios problemas a integridade do sistema. Suponha que uma aplicação atualize um arquivo em disco. O programa, por si só, não pode especificar diretamente as instruções que acessam seus dados no disco. Como o disco é um recurso compartilhado, sua utilização deverá ser gerenciada unicamente pelo sistema operacional, evitando que a aplicação possa ter acesso a qualquer área do disco indiscriminadamente, o que poderia comprometer a segurança e integridade do sistema de arquivos.

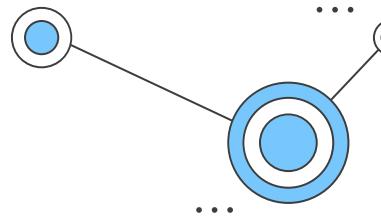
Como visto, fica claro que existem certas instruções que só devem ser executadas pelo sistema operacional ou sob sua supervisão, impedindo, assim, a ocorrência de problemas de segurança e integridade do sistema. As instâncias que tem o poder de comprometer o sistema são conhecidas como **instruções privilegiadas**, enquanto as **instruções não - privilegiadas** são as que não oferecem risco ao sistema.

# Modos de Acesso

A melhor maneira de controlar o acesso às instruções privilegiadas é permitir que apenas o sistema operacional tenha acesso a elas. Sempre que uma aplicação necessita executar uma instrução privilegiada, a solicitação deve ser realizada através de uma chamada a uma system call, que altera o modo de acesso do processador do modo usuário para o modo kernel. Ao término da execução da rotina do sistema, o modo de acesso retorna para o modo usuário. Caso uma aplicação tente executar uma instrução privilegiada diretamente em modo usuário, o processador sinalizara um erro, uma exceção é gerada e a execução do programa é interrompida.

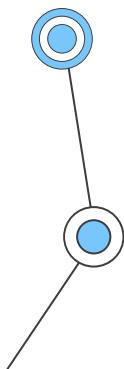


# Modos de Acesso



Utilizando o mesmo problema do acesso ao disco apresentado, para o programa atualizar um arquivo em disco a aplicação deve solicitar a operação de E/S ao sistema operacional por meio de uma system call, que altera o modo de acesso do processador de usuário para kernel. Após executar a rotina de leitura, o modo de acesso volta ao estado usuário para continuar a execução do programa.

O mecanismo de modos de acesso também é uma boa forma de proteger o próprio núcleo do sistema residente na memória principal. Suponha que uma aplicação tenha acesso a áreas de memória onde está o sistema operacional. Qualquer programador mal-intencionado ou um erro de programação poderia gravar nesta área, violando o sistema. Com o mecanismo de modos de acesso, para uma aplicação escrever numa área onde resida o sistema operacional o programa deve estar sendo executado no modo kernel.

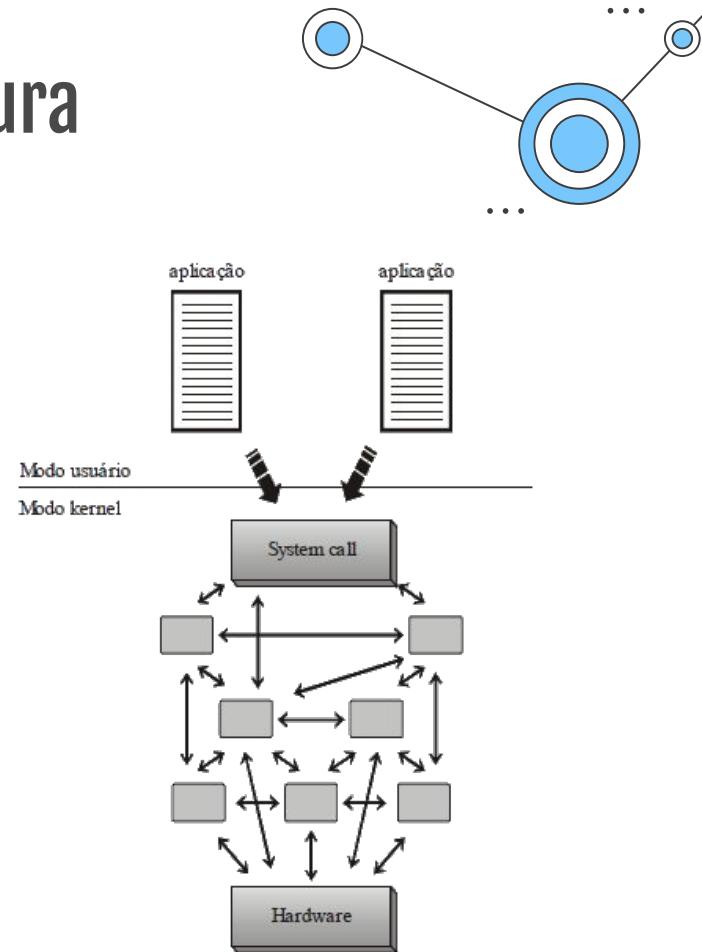


# Tipos de Arquitetura

## □ Arquitetura Monolítica

A **arquitetura monolítica** pode ser comparada com uma aplicação formada por vários módulos que são compilados separadamente e depois linkados, formando um grande e único programa executável, onde os módulos podem interagir livremente. Os primeiros sistemas operacionais foram desenvolvidos com base neste modelo, o que tornava seu desenvolvimento e, principalmente, sua manutenção bastante difíceis.

Devido a sua simplicidade e bom desempenho, a estrutura monolítica foi adotada no projeto do MS-DOS e nos primeiros sistemas Unix.



# Tipos de Arquitetura

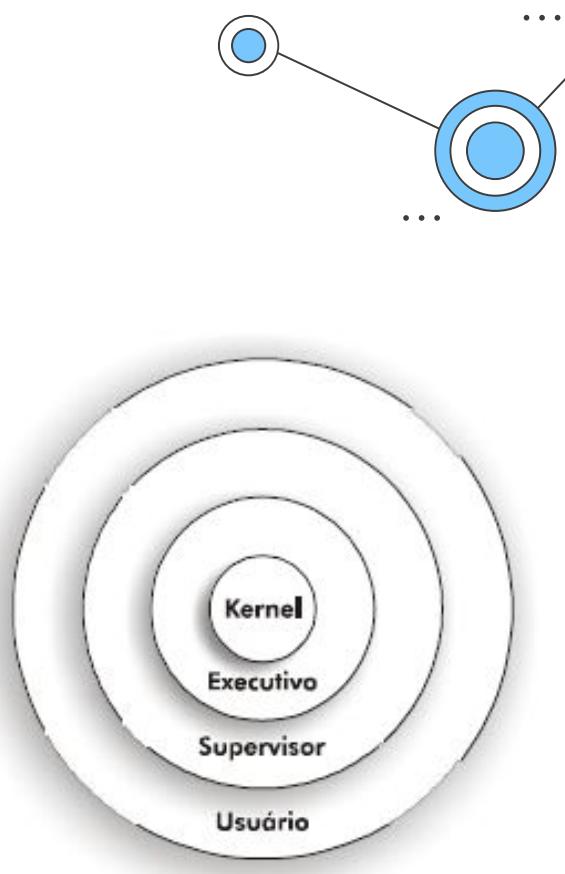
## □ Arquitetura Camadas

Com o aumento da complexidade e do tamanho do código dos sistemas operacionais, técnicas de programação estruturada e modular foram incorporadas ao seu projeto. Na **arquitetura de camadas**, o sistema é dividido em níveis sobrepostos.

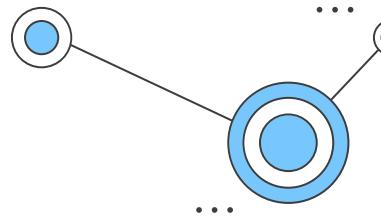
Cada camada oferece um conjunto de funções que podem ser utilizadas apenas pelas camadas superiores.

O primeiro sistema com base nesta abordagem foi o sistema THE (Technische Hogeschool Eindhoven), construído por **Dijkstra, na Holanda, em 1968**, e que utilizava seis camadas.

Posteriormente, os sistemas MULTICS e Open VMS também implementaram o conceito de camadas. Neste tipo de implementação, as camadas mais internas são mais privilegiadas que as mais externas.

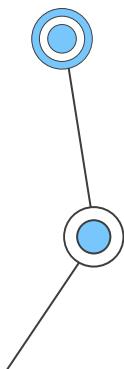


# Tipos de Arquitetura



A vantagem da estruturação em camadas é isolar as funções do sistema operacional, facilitando sua manutenção e depuração, além de criar uma hierarquia de níveis de modos de acesso, protegendo as camadas mais internas. Uma desvantagem para o modelo de camadas é o desempenho. Cada nova camada implica uma mudança no modo de acesso. Por exemplo, no caso do Open VMS, para se ter acesso aos serviços oferecidos pelo kernel é preciso passar por três camadas ou três mudanças no modo de acesso.

Atualmente, a maioria dos sistemas comerciais utiliza o modelo de duas camadas, onde existem os modos de acesso **usuário (não - privilegiado)** e **kernel (privilegiado)**. A maioria das versões do Unix e do Windows 2000 da Microsoft estão baseados neste modelo.



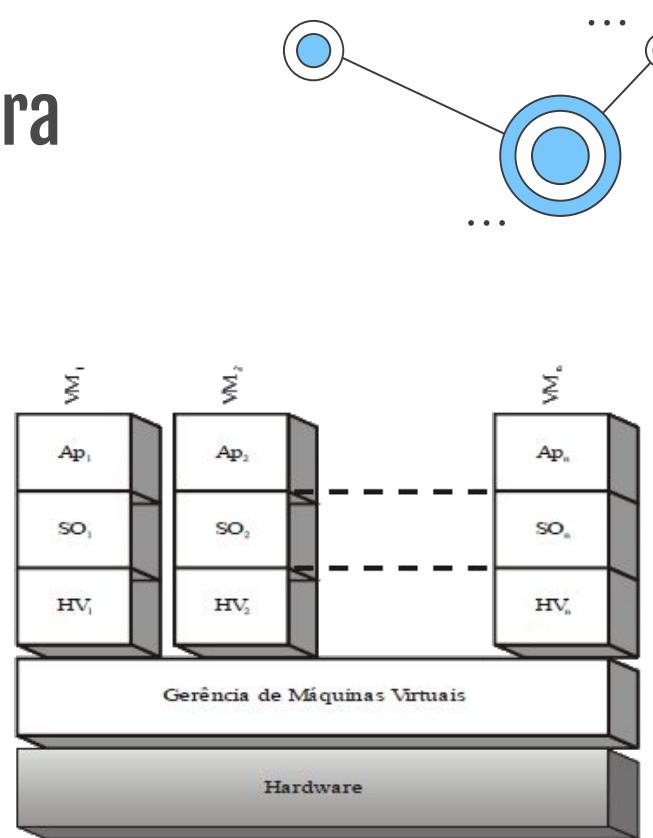
# Tipos de Arquitetura

## □ Maquina Virtual

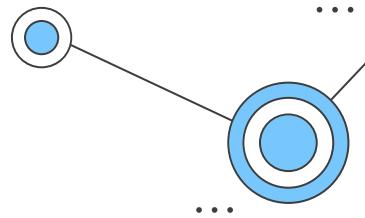
Um sistema computacional é formado por níveis, onde a camada de nível mais baixo é o hardware. Acima desta camada encontramos o sistema operacional, que oferece suporte para as aplicações, como visto na figura ao lado. O modelo de **maquina virtual**, ou **virtual machine (VM)**, cria um nível intermediário entre o hardware e o sistema operacional, denominado gerência de máquinas virtuais.

Este nível cria diversas máquinas virtuais independentes, onde cada uma oferece uma copia virtual do hardware, incluindo os modos de acesso, interrupções, dispositivos de E/S etc.

Como cada maquina virtual é independente das demais, é possível que cada VM tenha seu próprio sistema operacional e que seus usuários executem suas aplicações como se todo o computador estivesse dedicado a cada um deles. usuários e aplicações.

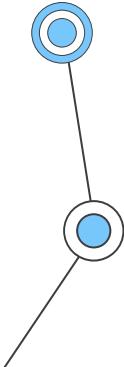


# Tipos de Arquitetura



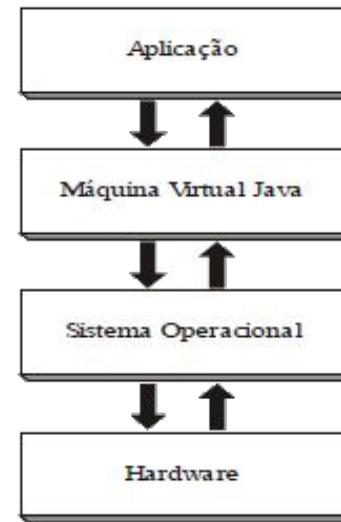
Como cada máquina virtual é independente das demais, é possível que cada VM tenha seu próprio sistema operacional e que seus usuários executem suas aplicações como se todo o computador estivesse dedicado a cada um deles. usuários e aplicações. Na década de 1960, foi implementado este modelo, permitindo que aplicações batch, originadas de antigos sistemas OS/360, e aplicações de tempo compartilhado pudessem conviver na mesma máquina de forma transparente a seus usuários e aplicações.

Além de permitir a convivência de sistemas operacionais diferentes no mesmo computador, este modelo cria o isolamento total entre cada VM, oferecendo grande segurança para cada máquina virtual. Se, por exemplo, uma VM executar uma aplicação que comprometa o funcionamento do seu sistema operacional, as demais máquinas virtuais não sofrerão qualquer problema. A desvantagem desta arquitetura é a sua grande complexidade, devido à necessidade de se compartilhar e gerenciar os recursos do hardware entre as diversas VMs.

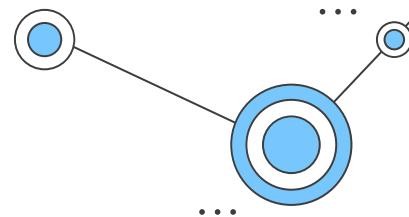


# Tipos de Arquitetura

Outro exemplo de utilização desta arquitetura ocorre na linguagem Java, desenvolvida pela Sun Microsystems. Para se executar um programa em Java é necessário uma máquina virtual Java (Java Virtual Machine—JVM). Qualquer sistema operacional pode suportar uma aplicação Java, desde que exista uma JVM desenvolvida para ele. Desta forma, a aplicação não precisa ser recompilada para cada sistema computacional, tornando-se independente do hardware e sistema operacional utilizá-lo. A desvantagem deste modelo é o seu menor desempenho se compara-lo a uma aplicação compilada e executada diretamente em uma arquitetura específica.



# Tipos de Arquitetura



## □ Arquitetura Microkernel

Uma tendência nos sistemas operacionais modernos é tornar o núcleo do sistema operacional o menor e mais simples possível. Para implementar esta ideia, os serviços do sistema são disponibilizados através de processos, onde cada um é responsável por oferecer um conjunto específico de funções, como gerência de arquivos, gerência de processos, gerência de memória e escalonamento.

Sempre que uma aplicação deseja algum serviço, é realizada uma solicitação ao processo responsável. Neste caso, a aplicação que solicita o serviço é chamada de cliente, enquanto o processo que responde a solicitação é chamado de servidor. Um cliente, que pode ser uma aplicação de um usuário ou um outro componente do sistema operacional, solicita um serviço enviando uma mensagem para o servidor. O servidor responde ao cliente através de uma outra mensagem. A principal função do núcleo é realizar a comunicação, ou seja, a troca de mensagens entre cliente e servidor.

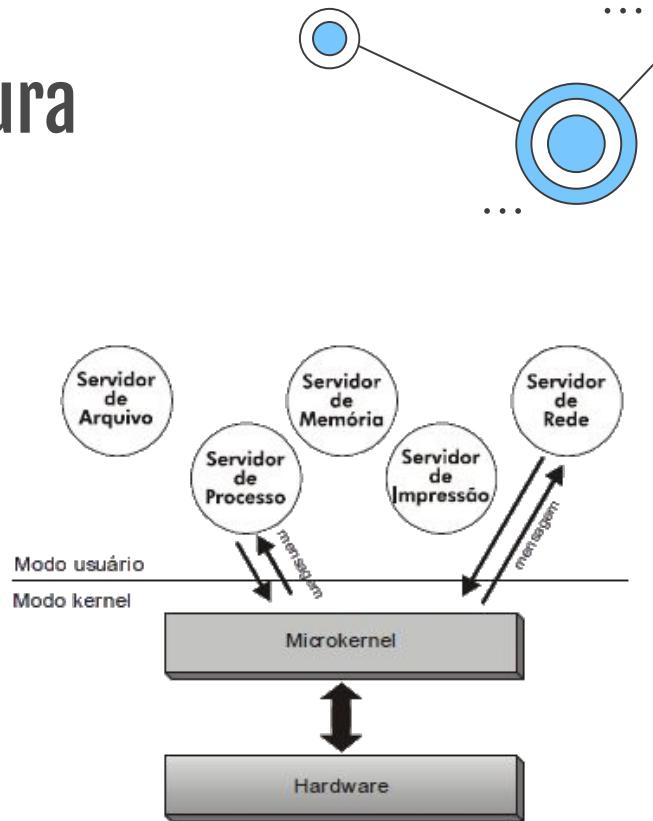
O conceito de arquitetura **microkernel** surgiu no sistema operacional Mach, na década de 1980, na Universidade Carnegie-Mellon. O núcleo do sistema Mach oferece basicamente quatro serviços: gerência de processos, gerência de memória, comunicação por troca de mensagens e operações de E/S, todos em modo usuário.



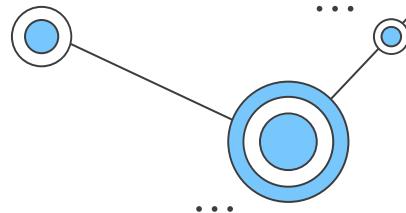
# Tipos de Arquitetura

A utilização deste modelo permite que os servidores executem em modo usuário, ou seja, não tenham acesso direto a certos componentes do sistema. Apenas o núcleo do sistema, responsável pela comunicação entre clientes e servidores, executa no modo kernel. Como consequência, se ocorrer um erro em um servidor, este poderá parar, mas o sistema não ficará inteiramente comprometido, aumentando assim a sua disponibilidade.

Como os servidores se comunicam através de trocas de mensagens, não importa se os clientes e servidores são processados em um sistema com um único processador, com múltiplos processadores (fortemente acoplado) ou ainda em um ambiente de sistema distribuído (fracamente acoplado). A implementação de sistemas microkernel em ambientes distribuídos permite que um cliente solicite um serviço e a resposta seja processada remotamente. Esta característica permite acrescentar novos servidores à medida que o número de clientes aumenta, conferindo uma grande estabilidade ao sistema operacional.



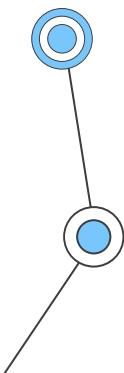
# Tipos de Arquitetura



Além disso, a arquitetura microkernel permite isolar as funções do sistema operacional por diversos processos servidores pequenos e dedicados a serviços específicos, tornando o núcleo menor, mais fácil de depurar e, consequentemente, aumentando sua confiabilidade. Na arquitetura microkernel, o sistema operacional passa a ser de mais fácil manutenção, flexível e de maior portabilidade.

Apesar de todas as vantagens deste modelo, sua implementação, na prática, é muito difícil. Primeiro existe o problema de desempenho, devido à necessidade de mudança de modo de acesso a cada comunicação entre clientes e servidores. Outro problema é que certas funções do sistema operacional exigem acesso direto ao hardware, como operações de E/S. Na realidade, o que é implementado mais usualmente é uma combinação do modelo de camadas com a arquitetura microkernel. O núcleo do sistema, além de ser responsável pela comunicação entre cliente e servidor, passa a incorporar outras funções críticas do sistema, como escalonamento, tratamento de interrupções e gerencia de dispositivos.

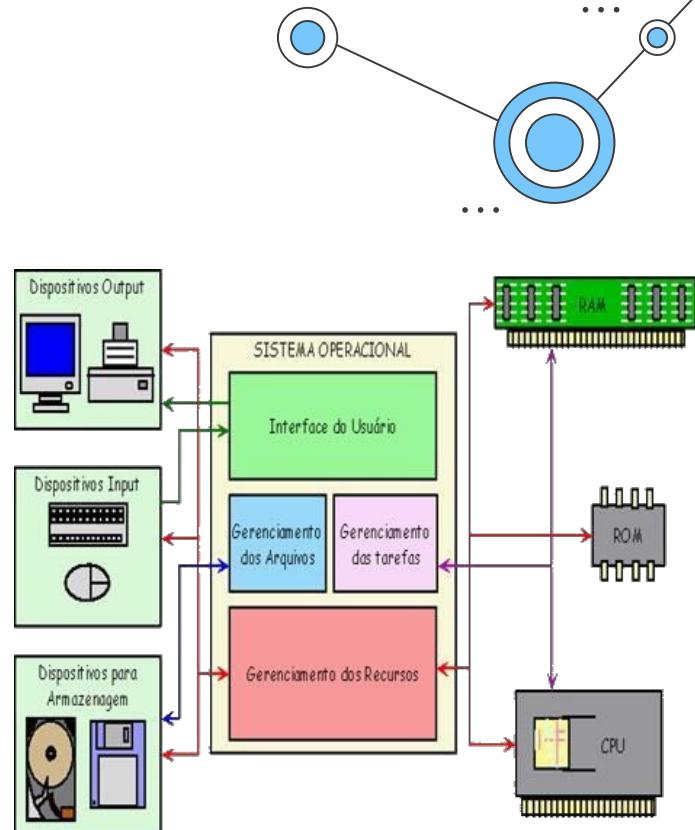
A maioria das iniciativas nesta área está relacionada ao desenvolvimento de sistemas Operacionais distribuídos.

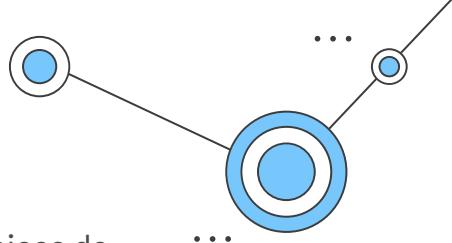


# Projeto do S.O

O projeto de um sistema operacional é bastante complexo e deve atender a diversos requisitos, algumas vezes conflitantes, como confiabilidade, portabilidade, manutenibilidade, flexibilidade e desempenho. O projeto do sistema irá depender muito da arquitetura do hardware a ser utilizado e do tipo de sistema que se deseja construir batch, tempo compartilhado, monousuário ou multiusuário, tempo real etc.

Os primeiros sistemas Operacionais foram desenvolvidos integralmente em assembly e o código possuía cerca de um milhão de instruções (IBM OS/360). Com a evolução dos sistemas e o aumento do número de linhas de código para algo perto de 20 milhões (MULTICS), técnicas de programação modular foram incorporadas ao projeto, além de linguagens de alto nível, como PL/I e Algol. Nos sistemas Operacionais atuais, o numero de linhas de código pode chegar a mais de 40 milhões (Windows 2000), sendo grande parte do código escrita em linguagem C/C++, utilizando em alguns casos programação orientada a objetos.





# Projeto do S.O

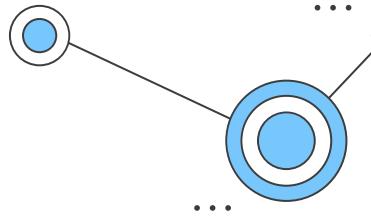
Uma tendência no projeto de sistemas operacionais modernos é a utilização de técnicas de orientação a objetos, o que leva para o projeto do núcleo do sistema todas as vantagens deste modelo de desenvolvimento de software. Existe uma série de vantagens na utilização de programação orientada a objetos no projeto e na implementação de sistemas operacionais.

## A seguir, os principais benefícios são apresentados:

- Melhoria na organização das funções e recursos do sistema;
- Redução no tempo de desenvolvimento;
- Maior facilidade na manutenção e extensão do sistema;
- Facilidade de implementação do modelo de computação distribuída.

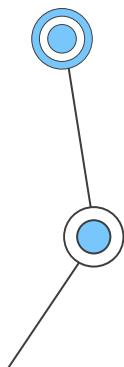
Além de facilitar o desenvolvimento e a manutenção do sistema, a utilização de linguagens de alto nível permite maior portabilidade, ou seja, que o sistema operacional seja facilmente alterado em outra arquitetura de hardware. Uma desvantagem do uso de linguagens de alto nível em relação a programação assembly é a perda de desempenho. Por isto, as partes críticas do sistema, como os device drivers, o escalonador e as rotinas de tratamento de interrupções, são desenvolvidas em assembly.

# Conclusão



De maneira mais objetiva, o sistema operacional refere-se a um ou mais softwares que tem como papel central gerenciar e ainda administrar todos os recursos presentes em um sistema. Isso envolve desde os componentes do hardware, sistemas de arquivos e até mesmo programas de terceiros e tudo isso é feito sem que o usuário perceba e já dizia Linus Torvalds criador do kernel Linux “A Principal Tarefa de um Sistema Operacional é você percebe que ele não Existe”.

Portanto, o sistema operacional faz a interface entre você e seu computador ou notebook. Mas, é bom lembrar que quando falamos em um dispositivo, podemos também nos referir a um celular ou tablet e até mesmo um console de videogame.

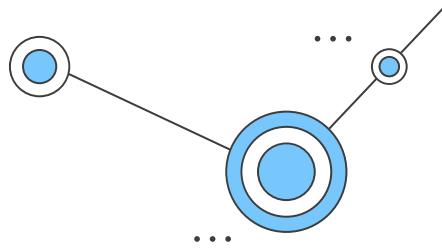


# 02

## Linguagens de Programação

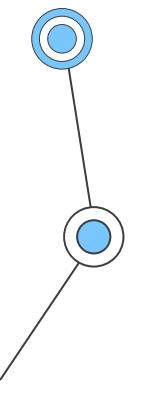
Um meio de comunicação entre  
computadores e humanos.

# Introdução



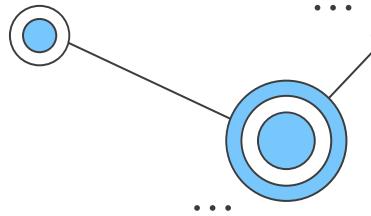
O C# é uma linguagem de programação moderna, robusta e orientada a objetos que está em constante evolução e adoção de profissionais no mundo inteiro. Com essa linguagem de programação é possível criar softwares para muitos tipos de aplicativos tais como celulares, computadores pessoais, jogos e muito mais!

O C# tem raízes na família de linguagens C e os programadores que já conhecem C ou C++ terão facilidade em migrar ou se atualizar para esse novo paradigma de programação. Por outro lado, para aqueles que nunca tiveram contato com a linguagem, a adoção e curva de aprendizado também é facilitada pela enorme comunidade técnica que a utiliza, pelos diversos sites e cursos que estão disponíveis.



Originalmente a linguagem foi criada pela Microsoft e tinha como objetivo fornecer alternativa ao mercado de software para criação de novos sistemas voltados para a plataforma Windows, porém, ao longo do tempo a linguagem foi evoluindo e conquistando mais adeptos não apenas do mercado corporativo, mas também do mundo do código-aberto. Atualmente é possível criar sistemas que rodam em Windows, Linux e Mac com a mesma linguagem de programação.

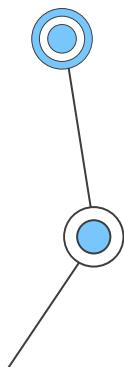
# Introdução

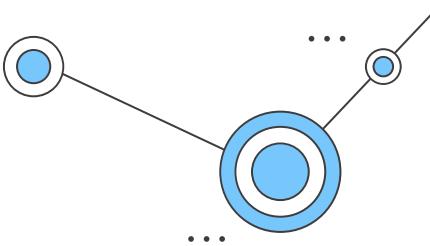


Como já vimos anteriormente, o C# é uma linguagem de programação orientada a objetos e orientada a componentes. Isso significa que a linguagem suporta construções e implementações de forma nativa para que nós como pessoas programadoras consigamos trabalhar nos projetos de uma forma clara e objetiva com recursos que são implementados na própria linguagem.

Por exemplo, se precisarmos construir um sistema que fará leituras de arquivos no formato XML, a própria linguagem de programação já nos provê implementações para que consigamos ler e manipular esse tipo de arquivo de forma fácil e intuitiva sem ter que implementar lógicas de programação para executar a mesma tarefa.

Desde a sua criação o C# adiciona com frequência recursos para dar suporte a novas cargas de trabalho e práticas de design de software. Com o C# podemos criar softwares nativamente para rodar em sistemas em nuvem, aplicativos para dispositivos móveis, sistemas embarcados, websites, aplicações para web, portais e muito mais.





# Modo de Compilação

Os arquivos binários são gerados durante o processo de compilação da nossa aplicação e antes de instalarmos estes binários nos ambientes, é importante entender como eles devem ser gerados e, para isto, existem dois modos de compilação: Debug e Release.

## **Modo de Debug**

Em Debug, nossa aplicação permite ser executada linha a linha, onde podemos depurar, ou executar o código linha por linha, parte por parte, a fim de resolvemos problemas e garantirmos que tudo está funcionando.

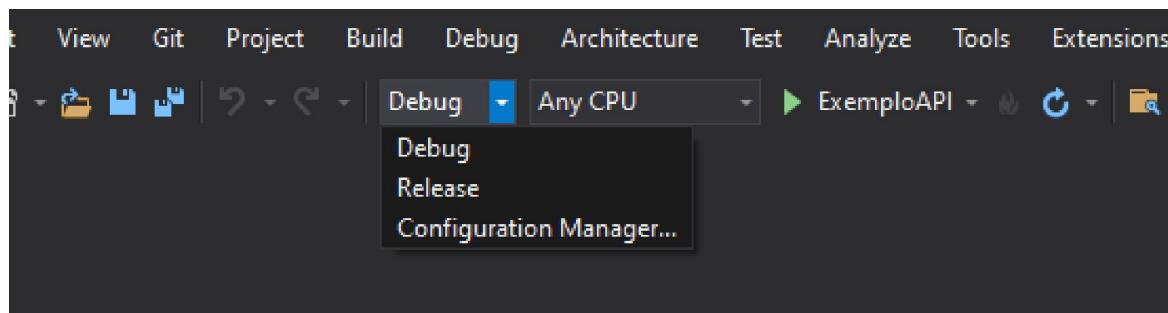
Neste modo, o Visual Studio, gera um arquivo extra, com a extensão .PDB (Program Database), que inclui informações sobre o código fonte. Isto permite também que façamos depuração em um ambiente sem o código fonte. Por exemplo, você tem uma DLL dentro do seu projeto e ao executar o Debug, acessando as linhas do seu código, se depara com um código externo, dentro da DLL. Se esta DLL tiver o arquivo PDB, será possível navegar pelo código fonte que ela possui. O modo Debug é utilizado geralmente durante o desenvolvimento.

# Modo de Compilação

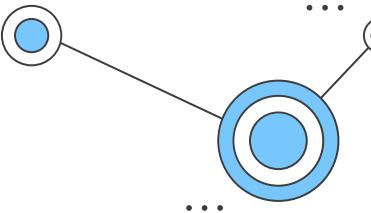
## ❑ Modo Release:

Este é o modo que utilizamos para colocar nosso código em produção, pois ele é mais otimizado pelo compilador, ou seja, não contém informações para Debug, somente execução. Sempre que for instalar seu software, compile no modo Release.

Dentro do Visual Studio, você pode escolher o modo de compilação no menu superior, conforme a figura a seguir:



# Namespace

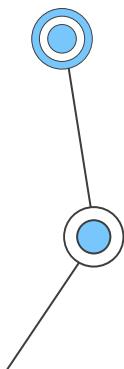


Os namespaces são usados no C# para organizar e prover um nível de separação de código fonte. Podemos considerá-lo como um “container” que consiste de outros namespaces, classes, etc.

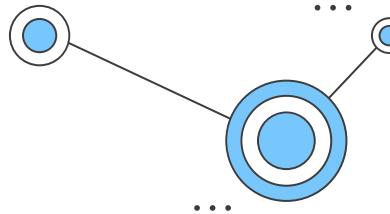
**Normalmente um namespace pode conter os seguintes membros:**

- Classes
- Interfaces
- Estruturas
- Delegates

Vamos entender o conceito de namespaces em um cenário um pouco mais real. Imagine que temos um grande número de arquivos e pastas em nosso computador. Imagine então como seria difícil gerenciar esses arquivos e pastas que foram colocados no mesmo diretório. Por esse motivo seria muito melhor se dividíssemos os arquivos em pastas diferentes onde faz sentido eles estarem, correto? A mesma analogia pode ser aplicada a utilização de namespaces.



# Namespace



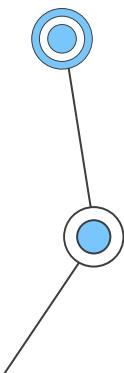
Os membros de um namespace podem ser acessados através de um operador de ponto (.). A sintaxe para acessar o membro dentro do namespace é **NomeDoNameSpace.NomeDoMembro**.

A principal funcionalidade do namespace é de fato organizar o projeto. A medida que ele vai ficando maior e com mais arquivos é extremamente importante que saibamos como segregar o projeto visando sobre a responsabilidade de cada componente e determinando suas ações de forma isolada. Uma boa prática recomendada pela Microsoft e diversos profissionais experientes é criar a estrutura do seu projeto seguindo a seguinte sintaxe:

**NomeDaEmpresa.NomeDoProjeto.CamadaDoProjeto.Funcionalidade**

**Exemplo:**

**Microsoft.LivroCSharp.CamadaDeDados.ConectorSqlServer**



# Variáveis

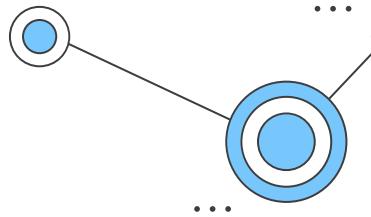
As variáveis são espaços na memória do computador onde podemos salvar, manipular e exibir informações.

As variáveis têm uma importância enorme no mundo de desenvolvimento de software pois são nelas que guardamos, manipulamos, alteramos e exibimos valores para os usuários finais ou simplesmente fazemos cálculos e manipulações e enviamos talvez para outros sistemas que queiram se comunicar.

De forma objetiva, a variável é meramente um nome dado para um armazenamento de dados que nosso programa vai manipular. Cada variável no C# possui um tipo específico que determina seu tamanho e o quanto de informação ela pode salvar na memória. Os tipos básicos são categorizados em:

Tipo	Exemplo
Tipos inteiros	sbyte, byte, short, ushort, int, uint, long, ulong, and char
Tipos de ponto flutuante	float and double
Tipos decimais	decimal
Tipos booleanos	true or false values, as assigned
Tipos nulos	Nullable data types

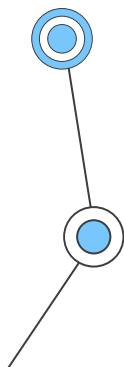
# Funções Internas



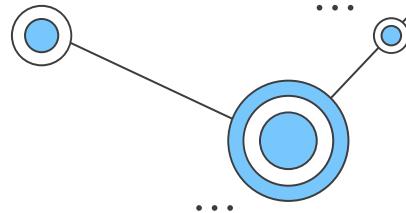
O C# contém diversos métodos internos da linguagem que permitem manipular textos, datas, fazer contas com números, e isto usamos em diversas ocasiões no dia a dia. Tais funcionalidades existem para facilitar a vida do desenvolvedor, senão seria preciso construir uma lógica, ocasionando possibilidades de erros.

## □ Funções de Textos

As funções de textos, como o próprio nome diz, são usadas para se manipular **strings**, cadeias de textos, pois nem sempre teremos o conteúdo limpo ou do jeito que precisamos. Com isto, podemos fazer o que quiser, limpar espaços em branco, obter a quantidade de caracteres, transformar tudo para maiúscula ou minúscula, trocar o conteúdo, enfim, depende do contexto.



# Funções de Texto

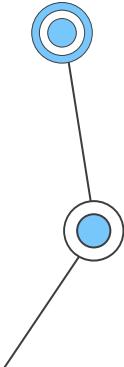


## Trim

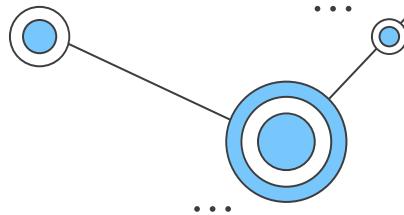
Este método retira todos os espaços em branco antes e depois de uma expressão. Caso tiver espaços em branco no meio da expressão, teremos que construir um código para retirar, não é o caso do **TRIM**. Este método é usado em interações com usuários, normalmente em cadastros onde são digitadas muitas informações em uma tela. Ou ainda, em casos de trazer informações de outra fonte de dados, arquivos textos, CSV, Excel, dados exportados de bancos de dados, enfim, o TRIM nos ajuda muito na consistência de capturar apenas a expressão em si.

## Length

O método **Length** conta a quantidade de caracteres da expressão, incluindo os espaços em branco em toda a expressão. É muito usado quando precisamos definir um tamanho limite que caiba num campo ou espaço a ser impresso ou mostrado. Em aplicativos onde a interação com usuário é feita, usamos para checar a quantidade de caracteres digitados, em importação de dados de outras fontes, checamos se o tamanho da expressão está de acordo como necessário, e caso seja preciso podemos até descartar o restante, por exemplo, em um endereço completo, aplicamos uma regra de no máximo 50 caracteres. Caso a expressão tenha 60, 70 caracteres, por exemplo, pegamos apenas os 50 primeiros. Enfim, há diversos casos de uso do Length.



# Funções de Texto



## **ToUpper**

O método **ToUpper** tem a função de transformar toda a expressão em letras maiúsculas. É usado em casos de formatação, onde precisamos chamar a atenção como nome, cargo, informações críticas, etc. O uso é simples, basta informar a cadeia de caracteres ou variável, seguido do método **ToUpper()**.

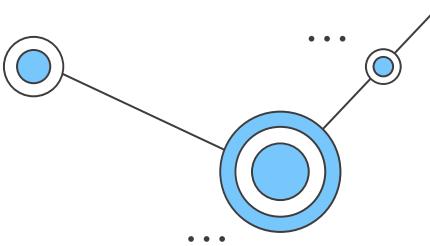
## **ToLower**

O método **ToLower** faz o contrário do **ToUpper**, ou seja, transforma toda a expressão em letras minúsculas. Se aplica também à expressões de caracteres ou variáveis. O uso se dá em casos de ler informações de qualquer fonte e formatá-la para minúsculo.

## **Replace**

O método **Replace** é usado para substituir cadeias de expressões de textos. Existem dois parâmetros, o primeiro indica qual o texto a ser substituído e o segundo é o texto que substituirá. Um uso comum é em casos de tratamento de dados, por exemplo, numa lista de endereços, trocar os termos "R.", "RUA" ou "Street" por "Rua".





# Funções de Texto

## □ Split

O método **Split** é muito usado em situações de tratamento de dados oriundos de arquivos textos, CSV, Excel, onde é preciso separar as cadeias de caracteres através de um caractere chave. O mais usado é o espaço em branco, mas pode ser qualquer caractere.

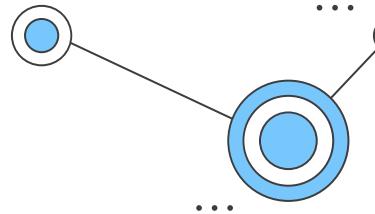
## □ Substring

O método **Substring** é usado para extrair parte do texto a partir de uma posição inicial. O tamanho do texto a ser capturado pode ou não ser informada, caso não seja, todo o texto a partir da posição é capturado. Caso declarado o tamanho, é capturado o texto conforme o número de caracteres. A sintaxe `Livro.Substring(5, 14)` informa que o texto a ser capturado deverá iniciar na posição 5 e pegar os próximos 14 caracteres.

## □ IsNullOrEmpty

O método **IsNullOrEmpty** verifica se uma **String** está nula ou vazia. Isto é muito usado onde há interação com o usuário, afim de consistência de dados.

# Funções de Datas



As funções de datas permitem manipular qualquer informação de uma data que esteja no modelo **DateTime** contendo o dia, o mês e o ano. Sendo assim, conseguimos fazer contas com datas, adicionar ou subtrair, dias, meses e anos, aplicar formatações customizadas, obter a data e hora completa do sistema operacional, converter em texto, ler um texto e converter em data, entre outras.

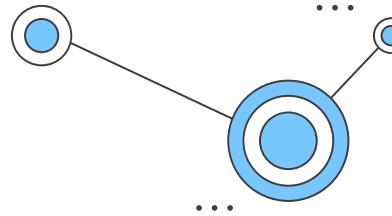
## **DateTime**

Para o C# uma data, hora, minuto, segundos e ticks dependem do tipo **DateTime**. Todos estes são conhecidos, exceto o **Ticks**, que é um tipo **Long** que permite atribuir um número para **milissegundos** e **nanosegundos**.

O **DateTime** para uma data comum é composto pelo dia (1-31), mês (1-12) e o ano, todos valores numéricos. A ordem de entrada sempre será o Ano, Mês, Dia, nesta ordem respectivamente.



# Funções de Datas



## Today

A propriedade **Today** do **DateTime** retorna a data completa com o dia, mês e ano do sistema operacional. Usamos com muita frequência em aplicações onde precisamos saber a data no dia de hoje, assim o usuário não tem como informar, a data é capturada diretamente da máquina e não existem parâmetros.

## Now

A propriedade **Now** do **DateTime** retorna a data, hora, minutos e segundos automaticamente do sistema operacional. Em aplicações de bolsa de valores, transações em bancos e uso de medicamentos são apenas alguns exemplos de onde são utilizadas.

## Day/Month/Year

A partir de uma data completa é possível extrair as devidas partes como dia, mês e ano, armazenar em variáveis, fazer contas, manipular, enfim, fazer o que quiser conforme o escopo da aplicação. Para isto, basta criar uma data completa e usar as propriedades **Day**, **Month** e **Year** do objeto **DateTime**.



# Funções de Datas

## □ Manipular Data

Em uma data **DateTime** válida podemos adicionar dias **AddDays(n)**, meses **AddMonths(n)** e anos **AddYears(n)**, basta informar o respectivo número a ser adicionado.

Vamos a um exemplo clássico de pedido de compra. No código a seguir temos a data do pedido (**dtPedido**) que captura o dia atual (**Today**), a data de vencimento (**dtVencto**) que são adicionados 35 dias à **dtPedido** (**AddDays(35)**), a data do pagamento (**dtPagto**) que são adicionados 2 meses à **dtVencto (AddMonths(2))**. Em seguida, as datas do pedido e vencimento são exibidas com o formato customizado **dd/MM/yyyy**. E como o C# dispõe de duas formatações prontas, vamos usá-las em **dtVencto** com o formato longo (**ToLongDateString**) e o formato curto (**ToShortDateString**). Toda data tem obrigatoriamente o dia da semana (domingo, segunda, ..., sábado), e para saber qual é o dia usamos a propriedade **DayOfWeek**, o qual está sendo aplicada à **dtVencto**.

Podemos usar um formato específico do **ToString** de acordo com a cultura definida, neste caso a cultura do Brasil pt-BR, **dtVencto.ToString("dddd", new CultureInfo("pt-BR"))**. Caso use uma cultura é preciso adicionar o **using System.Globalization**. Já o dia do ano (**DayOfYear**) aplicado à **dtVencto** mostra quantos dias foram corridos desde o início do ano. E, para saber quantos dias se passaram entre duas datas, usamos o **Subtract**, onde referenciamos a maior data, neste caso **dtPagto**, seguido do método **Subtract**, passando como parâmetro a data a ser comparada, **dtPedido**. Isto retorna o número de dias corridos.

# Funções de Datas

```
using System;
using System.Globalization;

static void Main(string[] args)
{
    DateTime dtPedido = DateTime.Today;

    // adiciona 35 dias
    DateTime dtVencto = dtPedido.AddDays(35);

    // adicionar 2 meses
    DateTime dtPagto = dtVencto.AddMonths(2);
    Console.WriteLine($"Pedido feito em {dtPedido:dd/MMM/yyyy} vence em {dtVencto:dd/MMM/yyyy}");
    Console.WriteLine($"Formatação completa: {dtVencto.ToString("yyyy-MM-dd HH:mm:ss")}");
    Console.WriteLine($"Formatação curta: {dtVencto.ToShortDateString()}");

    // dia da semana
    Console.WriteLine($"dia da semana: {dtVencto.DayOfWeek}");
    Console.WriteLine($"dia do semana em português: {dtVencto.ToString("ddd", new CultureInfo("pt-BR"))}");
    Console.WriteLine($"Número do dia da semana: {(int)dtVencto.DayOfWeek}");

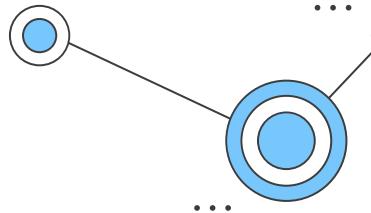
    // dia do ano
    Console.WriteLine($"dia do ano: {dtVencto.DayOfYear}");

    // subtrai 2 datas
    var qtdeDias = dtPagto.Subtract(dtPedido);
    Console.WriteLine($"Entre o pedido e o pagamento foram {qtdeDias:dd} dias");
}
```

# Funções de Datas

```
# Resultado:  
**Pedido feito em** *06/abr/2021* **vence em** *11/mai/2021*  
**Formatação completa:** *terça-feira, 11 de maio de 2021*  
**Formatação curta:** *11/05/2021*  
**dia da semana:** *Tuesday*  
**Número do dia da semana:** *2*  
**dia da semana em português:** *terça-feira*  
**dia do ano:** *131*  
**Entre o pedido e o pagamento foram** *96* **dias**
```

# Conversões de Dados

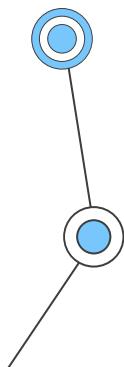


No C# temos dois tipos de dados que são sempre armazenados na memória, sendo tipos de **valor** e **referência**. Quando atribuímos um valor a uma variável dos tipos **int**, **float**, **double**, **decimal**, **bool** e **char** são do tipo **VALOR**. Isto porque o conteúdo vai diretamente para um local na memória.

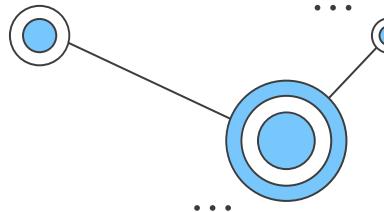
Já o tipo por **REFERÊNCIA**, armazena o endereço do valor onde está armazenado, por exemplo, **object**, **string** e **array**. Em qualquer tipo de aplicação é comum a conversão de tipos de dados, **int** para **double**, texto para data, **object** para **float** e vice-versa.

A estas conversões chamamos de **Boxing** e **Unboxing**. **Boxing** é a conversão de um tipo de valor para o tipo de objeto ou qualquer tipo de interface implementado por este tipo de valor. O **boxing** está **implícito**.

```
// boxing
int percentual = 10;
object objeto1 = percentual;
```



# Conversões de Dados

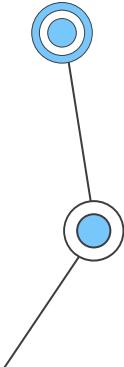


**Unboxing** é o inverso do **Boxing**. É a conversão de um tipo de referência em tipo de valor. O **unboxing** extrai o valor do tipo de referência e atribui a um tipo de valor. O **unboxing** é explícito, ou seja, precisamos declarar, por exemplo **(int) objeto2**.

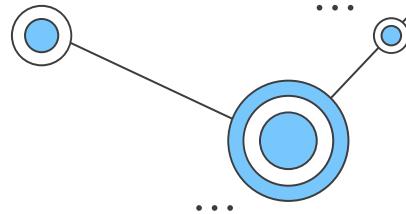
```
// unboxing
object objeto2 = 10;
int desconto = (int)objeto2;
```

## Como os valores funcionam na memória do CLR (Common Language Runtime)?

Todas as informações são armazenadas na memória quando atribuímos valores aos objetos. O valor e o tipo de dado é apenas uma referência na memória. No exemplo acima, **int percentual** é atribuído ao **object objeto1**, sendo que **objeto1** é apenas um endereço e não um valor em si. Com isto, o CLR configura o tipo de valor criando um novo **System.Object** no **heap (área da memória)** e atribui o valor de **percentual** a ele. Em seguida, atribui um endereço desse objeto ao **objeto1**. Isto é denominado **Boxing**.



# Conversões de Dados



## □ Parse

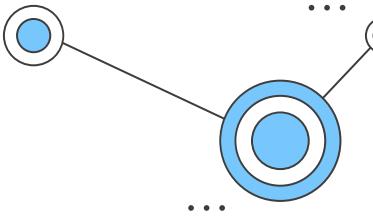
Para fazer uma conversão de tipos de dados é através do **Parse**. A sintaxe deve conter o tipo de campo (**Int32, Int64, int, DateTime**) seguido do **.Parse** contendo o valor a ser convertido EX: **Int16.Parse("150")**.

## □ ConvertTo

Há mais uma forma de conversão de tipos que usamos no C#, chama-se **Convert.To(tipo)**. A sintaxe é simples, basta escrever o **Convert. To+tipo** de dado, que pode ser **Int16, Int32, Int64, Decimal, String, Boolean ou DateTime** EX: **Convert.ToBoolean(0)**.



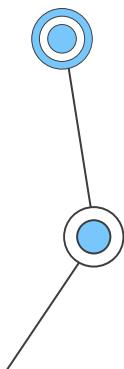
# Método de Extensão



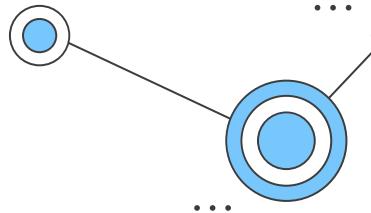
O C# permite ao desenvolvedor criar códigos próprios para auxiliar na produtividade e uso em qualquer parte do projeto, por exemplo, um método de extensão para auxiliar na formatação de um número, você cria uma vez um determinado formato customizado e o usa em todo o projeto, apenas chamando o método.

O conceito de método de extensão é criar um método customizado para um determinado tipo de dado (**int, DateTime, string, etc**) e aplicar uma ação. Os tipos primitivos do C# contém todas as possibilidades possíveis de customização, no entanto, de acordo com o tipo de aplicação é normal o desenvolvedor criar seus próprios formatos e funções, a fim de facilitar o uso e ter produtividade.

Para um método de extensão é regra obrigatória que a classe (**public static class**) e os métodos (**public static string**) sejam estáticos.



# Método de Extensão



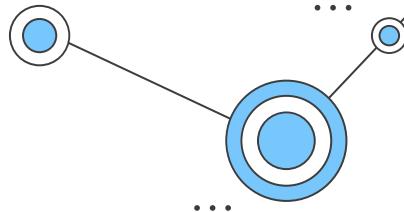
Em nosso exemplo o método **FormatarData**, note que na declaração ele retorna um texto (**string**), recebe como entrada um campo do tipo data **DateTime** e outro do tipo texto (**string**). Este método receberá uma data e aplicará o formato de acordo com o que o desenvolvedor informar na variável **formato**.

Note ainda que a variável **data** está declarada com o **THIS**, o que significa que pertence a esta classe, ou seja, é dela mesma. E qual o retorno? O retorno do método está após a lambda => no código **data.ToString(formato)**. Então, a esta data será aplicado através do método **ToString()** do C#, o formato que foi passado pelo desenvolvedor na variável **formato**.

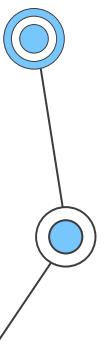
```
public static string FormatarData(this DateTime data, string formato) => data.ToString(formato);
```

```
WriteLine(DateTime.Today.FormatarData("dd/MMM/yyyy"));
/* Resultado: 05/jul/2021 */
```

# Delegate Func < >



**Delegate** nada mais é que um **ponteiro para uma função** e para utilizá-lo você só precisa respeitar o seu contrato que seria o tipo de retorno e parâmetros, caso tenha. O delegate também é conhecido por callback, porque ele pode ser passado como argumento para outra função e esse recurso é bem conhecido na linguagem javascript.



```
1 reference
class Pagamento
{
    1 reference
    public [delegate] void PagamentoProcessado(int numeroPedido);

    2 references
    public void Processar(PagamentoProcessado pagamentoProcessado)
    {
        for (int i = 1; i <= 10; i++)
        {
            // # Simulando Pagamento
            Thread.Sleep(1000);

            // # Status Processamento
            pagamentoProcessado(i);
        }
    }
}
```

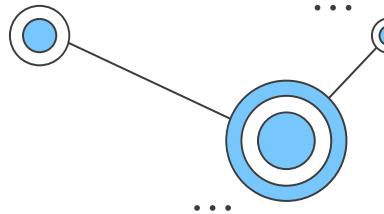
```
// # Callback
1 reference
static void ExibirStatusProcessamento(int numeroPedido)
{
    Console.WriteLine($"Pedido N° {numeroPedido} processado!");
}

0 references
public static void Main(string[] args)
{
    var pagamento = new Pagamento();

    // Exemplo 1
    pagamento.Processar(ExibirStatusProcessamento);

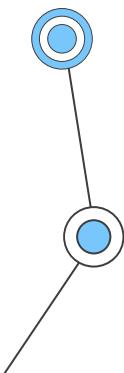
    // Exemplo 2
    pagamento.Processar(numeroPedido =>
    {
        Console.WriteLine($"Pedido N° {numeroPedido} processado!");
    });
}
```

# Delegate Func < >



O uso de **Delegates** é um dos melhores recursos da linguagem C#. Basicamente temos situações onde é preciso disparar um código que será executado em tempo de execução, de acordo com a situação. Pense num conjunto de métodos assinados e prontos para uso, só que a chamada será gerada em tempo de execução apenas. Você não terá a notação verbosa, clara, detalhando todos os procedimentos passo a passo, as coisas estarão encapsuladas num conjunto de instruções a serem geradas dinamicamente!

Parece confuso, então vou tentar explicar com um exemplo simples e funcional. Todos os proprietários de carros, quando é preciso fazer a revisão no mecânico, é preciso seguir um **checklist** de vários itens. Quando o mecânico conecta o computador ao carro para diagnosticar problemas no motor, existem diversos blocos de códigos (podemos entender como métodos) que podem ou não ser disparados de acordo com o problema. Por exemplo, checar pressão dos pneus, caso estiver baixo, é chamado o método para calibrar o pneu, o qual recebe como parâmetro o tipo de carro, características do pneu, etc, para então saber quanto de pressão calibrar.



# Delegate Func < >

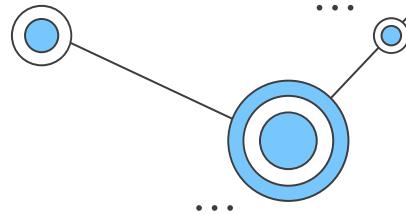


Já em casos de revisão do óleo do motor, é preciso disparar o código de verificar a viscosidade do óleo, a quantidade existente, a espessura do mesmo, etc e define se está na hora ou não de efetuar a troca. Caso esteja, é preciso disparar um outro código para checar o filtro do óleo, outro código para o filtro do ar condicionado. Enfim, note que dependendo do fluxo dos resultados, são disparados diversos blocos de códigos, rotinas, métodos, passando diversos parâmetros.

A estes blocos que existem e podem ou não ser disparados em tempo de execução é o que entendemos como **Delegates**. Eles estão sempre prontos para serem criados e executados em tempo de execução.

No C# temos o comando **FUNC** que é um **delegate** que tem até **16 entradas (parâmetros)** e apenas **uma saída (resultado)**.

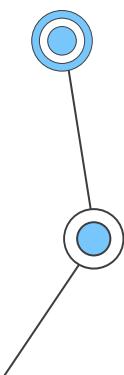
# Delegate Func <>



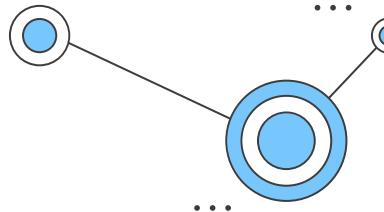
Em nosso exemplo vamos utilizar o **Delegate Func** de maneira que tenhamos 3 entradas de dados e uma saída, todos do tipo **decimal**. O que faremos é um código para calcular o Imposto, tendo como entradas os valores dos salários, o percentual de desconto e a alíquota. Existe uma condição que se o salário for menor ou igual a 1000, o Imposto é zero, caso contrário, é calculado o percentual e subtraída a alíquota.

Vamos por parte, primeiro temos o **Func<decimal, decimal, decimal, decimal>** com as quatro variáveis do tipo decimal. Em seguida, o nome do método Imposto, neste caso. E o código atribuído => se o salario <= 1000, o retorno é 0; caso contrário a fórmula: salario \* (perc / 100) - alíquota.

Já na chamada do **delegate Func Imposto** são passados os valores respectivamente.



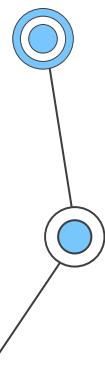
# Delegate Func ◊



```
Func<decimal, decimal, decimal, decimal> Imposto = (salario, perc, aliquota) =>
{
    return salario <= 1000 ? 0 : salario * (perc / 100) - aliquota;
};

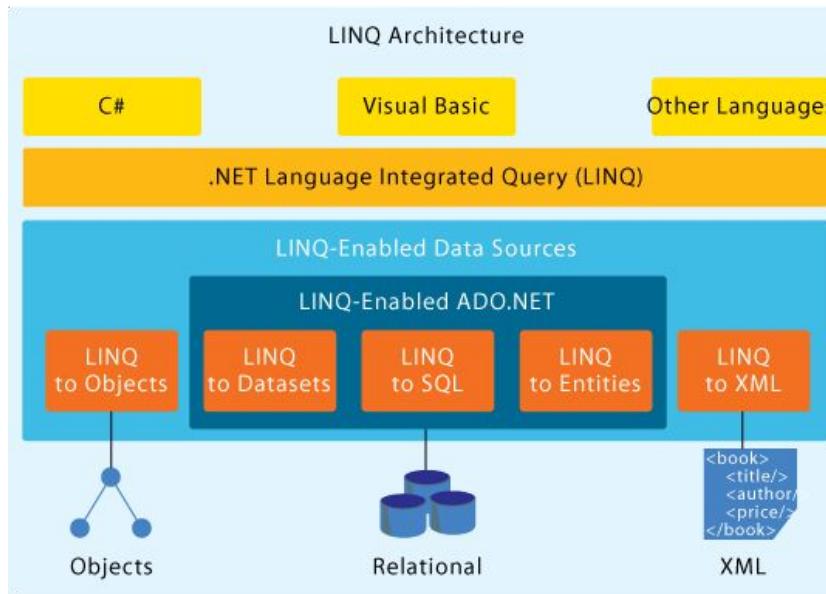
WriteLine("---- calculo do Imposto ---");
WriteLine(Imposto(1000, 10, 10));
WriteLine(Imposto(5000, 27.5M, 80));
WriteLine(Imposto(23500, 32.5M, 180));

/*
Resultado:
----- Cálculo do Imposto -----
0
1295,000
7457,500
*/
```

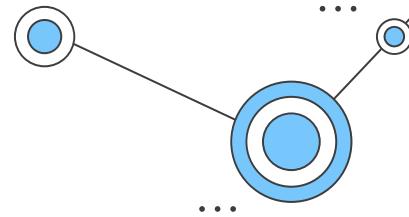


# LINQ (Language Integrated Query)

Agora vamos conhecer um pouco sobre **LINQ (Language Integrated Query - Consulta Integrada à Linguagem)**, esse recurso foi inicialmente adicionado à versão do .NET Framework 3.5 no final do ano de 2007 juntamente com o Visual Studio 2008. Esse foi um marco muito importante para o ecossistema .NET, e desde então, tem ganhado melhorias significativas.

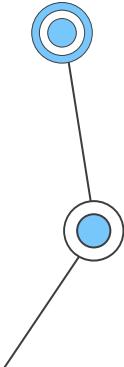


# LINQ (Language Integrated Query)

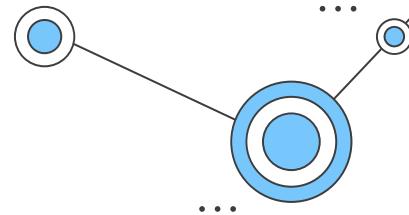


O LINQ surgiu para fornecer a capacidade de efetuar consultas em fonte de dados diferentes, seu principal objetivo é abstrair toda complexidade ao fazer consultas em uma fonte de dados, seja ela um XML, Banco de dados, ou até mesmo uma cadeia de caracteres. Antes do surgimento do LINQ, filtrar dados em uma fonte de dados como, por exemplo, um ARRAY, precisava de um certo conhecimento, assim era necessário criar um algoritmo que atendesse os critérios de seu filtro.

Consultar dados em diferentes fontes ficou muito mais fácil, dado que a complexidade abstraída pelo LINQ não nos obriga a criar consultas específicas para diferentes fontes de dados. Por exemplo, se você está escrevendo uma aplicação que se conecta a um banco de dados relacional, não necessariamente você precisa aprender **SQL (Linguagem de Consulta Estruturada)** para fazer suas consultas, da mesma forma para consultar dados em um XML, você não precisará ficar percorrendo todos os nós do documento, o LINQ já fornece a melhor experiência para fazer consultas, seja ela em objetos, XML ou para qualquer coleção de objeto que forneça à você suporte ao **IEnumerable** ou a interface genérica **IEnumerable<T>**.

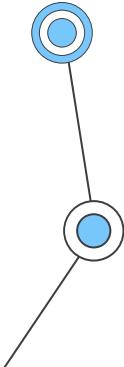


# LINQ (Language Integrated Query)

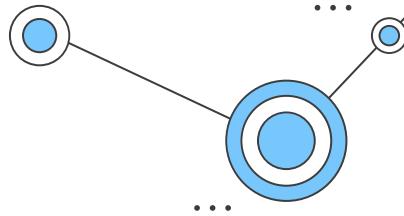


O LINQ aplica filtros em consultas por meio de expressões ou **delegates**, em tempo de desenvolvimento podemos explicitamente especificar quais são os filtros e critérios desejados para serem aplicados em sua consulta. Quando fazemos uma consulta sobre um **IEnumerable<T>**, significa que a consulta será compilada para um **delegate**, mas se fizermos uma consulta sobre um **IQueryable<T>**, ela será compilada para uma árvore de expressões. Isto possibilita que algum manipulador possa extrair informações de sua expressão e consiga tomar alguma decisão sobre o que fazer com o filtro utilizado em sua consulta. Um bom exemplo são os **ORM's (Modelos de Objetos Relacionais)**, como o popular **Entity Framework Core**.

Podemos escrever suas consultas LINQ de duas formas: a mais comum é a “sintaxe de consulta”, dado que é mais expressiva, muito mais próximo do que um DBA, por exemplo, poderia escrever para executar uma consulta em um banco de dados, “sintaxe de consulta” foi basicamente inspirada em uma linguagem natural, onde o próprio texto se explica e de fácil entendimento; podemos também fazer consultas por meio da “sintaxe de método”, como o próprio nome já diz, basicamente utiliza métodos para fazer consultas.

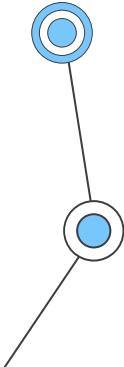


# LINQ (Language Integrated Query)



E qual a diferença? Em termos de performance não existe nenhuma diferença, dado que quando seu programa é compilado, ambas produzem o mesmo resultado, mas, existe uma bifurcação aqui para tomadas de decisões. Ao utilizar “sintaxe de consulta” realmente torna o código mais expressivo, mas existem algumas limitações ao utilizar esse tipo de consulta, por exemplo, algumas operações não conseguiremos fazer como Min, Max, Count, Sum, forçando mesclar a consulta utilizando a “sintaxe de método”. Sendo assim, é necessário conhecer melhor ambos tipos de consultas suportado pelo LINQ, mas já posso lhe garantir que usando “sintaxe de método” teremos muito mais flexibilidade.

O LINQ realmente nos proporciona uma excelente experiência ao consultar informações em uma fonte de dados, além de efetuar consultas, conseguimos manipular o resultado, ou seja, transformar os dados do resultado da consulta para entregar os dados formatados ao consumidor. Para ficar mais didático, pense no cenário onde fazemos uma consulta em um banco de dados e transformamos o resultado materializado de uma consulta em um XML, ou até mesmo uma string formatada para ser gravada em um arquivo CSV.



# LINQ (Language Integrated Query)

Antes de mergulhar profundamente no **LINQ** é preciso conhecer a diferença entre **IEnumerable<T>** e **IQueryable<T>**, isso vai ser importante para entender melhor o funcionamento dos filtros que serão aplicados nas consultas pelo **LINQ**. **IEnumerable<T>** tem como objetivo expor um enumerador para que possamos iterar sobre a instância de um objeto. Cabe ressaltar que todo processamento é executado na memória, os métodos que possuem sobrecargas aguardam um **delegate** do tipo **Func<T, TResult>** onde sempre é fornecido um tipo de entrada e um tipo de saída, conforme a figura a seguir.

```
var numeros = new[] ( 1, 2, 3 );

// Função (entrada inteiro, saída booleano)
Func<int, bool> funcao = i => i > 2;
var resultado = numeros.Where(funcao);

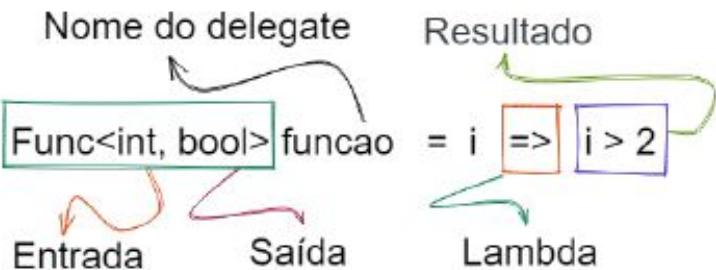
// Faz iteração na variável resultado
foreach (var numero in resultado)
{
    // Imprime número
    Console.WriteLine (numero) ;
}

// Resultado:  3
```

# LINQ (Language Integrated Query)

Vamos entender o que o código está fazendo. Primeiramente criamos uma coleção de números de 1 a 3, em seguida criamos uma função que valida se um número é maior que dois, por meio de uma expressão **lambda**, e assim aplicamos um filtro usando o método **where** o qual iremos abordar sobre ele logo mais.

Depois fizemos uma iteração sobre o enumerador da variável resultado e imprimimos o resultado no console da aplicação, veja uma ilustração na figura a seguir.



# LINQ (Language Integrated Query)

Não iremos nos aprofundar no **IQueryable<T>**, dado que nosso objetivo é apenas apresentar conceitos do LINQ, mas vamos entender um pouco sobre seu comportamento. O **IQueryable<T>** diferentemente do **IEnumerable<T>** espera em seus métodos expressões do tipo

**Expression<Func<T,TResult>>**, é dessa forma que os escritores de provedores que desejam fazer implementações de consultas são capaz de extrair da árvore de expressão os parâmetros suficientes para montar uma instrução SQL que possa ser executada em um banco de dados relacional, por exemplo, um provedor que faz bastante uso do **IQueryable<T>** é o famoso **ORM Entity Framework Core.**

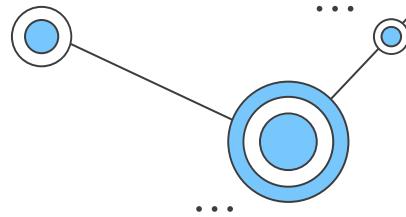
```
var numeros = new[] ( 1, 2, 3 ).AsQueryable();
// Expressão
Expression<Func<int,bool>> funcao = p => p > 2;

// Filtro
var resultado = numeros.Where(funcao) ;

// Faz iteração na variável resultado
foreach (var numero in resultado)
{
    // Imprime número
    Console.WriteLine(numero) ;
}

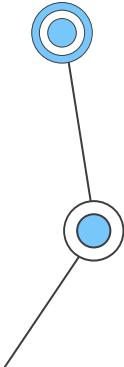
// Resultado: 3
```

# LINQ (Language Integrated Query)



Ao observarmos a figura anterior, a única diferença encontrada com uma consulta de uma coleção que herda o tipo **IEnumerable<T>**, é que ao invés de passar apenas um delegado (**delegate**), agora estamos passando uma expressão com um delegado (**delegate**). É exatamente por meio dessas expressões que os criadores de **ORM's** ou escritores de provedores conseguem identificar pela árvore de expressão que tipo de dados estamos consultando, quais operadores utilizamos e que tipo de retorno esperamos.

A seguir iremos falar um pouco mais sobre consultas **LINQ**, aquela que é processada apenas na memória, como já explicado anteriormente. Quando escrevemos uma consulta LINQ ela é executada automaticamente em tempo de execução? A resposta é não, ela será processada somente quando fizermos uma iteração por meio do **foreach** ou usar algum método de extensão como por exemplo o **ToList** ou **ToArrayList**.



# LINQ (Language Integrated Query)

Vamos colocar em prática toda teoria adquirida até aqui, sendo assim usaremos como um exemplo hipotético um array com dez números, onde precisamos filtrar os números que são maiores que cinco e imprimir o resultado na tela, conforme figura a seguir.

```
// Array de números
var numeros = new[] ( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 );

// Filtrar números
var numerosFiltrados = numeros.Where(n=> n > 5);

// Fazer iteração
foreach(var numero in numerosFiltrados)
    Console.WriteLine(numero);

/* Resultado: 6 7 9 10 */
```

# LINQ (Language Integrated Query)

Observamos como foi simples fazer uma consulta usando **LINQ** em uma fonte de dados. Novamente, para reforçar os conceitos, a consulta só foi processada exatamente no momento que foi feito a iteração sobre a variável **numerosFiltrados** por meio do **foreach**, e não ao usar o método de extensão **Where**. Também foi usado o estilo de consulta “**sintaxe de método**”, mas podemos usar “**sintaxe de consulta**” da qual já falamos no início deste capítulo, conforme figura a seguir, o resultado é o mesmo.

```
// Array de números
var numeros = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Filtrar números (sintaxe consulta)
var numerosFiltrados = from n in numeros where n > 5 select n;

// Fazer iteração
foreach (var numero in numerosFiltrados)
    Console.WriteLine (numero);

// Resultado: 6 7 8 9 10
```

# LINQ (Language Integrated Query)

## □ Operadores Suportados pelo LINQ

O LINQ possui inúmeros operadores de consultas, alguns são suportados apenas por meio de **"sintaxe de método"**, ou seja, métodos de extensões. Os operadores mais comuns são: projeção de dados, restrições, junções, ordenações, agrupamentos, agregações e paginações.

Métodos de projeção e restrição		
Select	SelectMany	Where

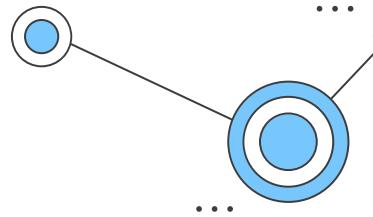
Métodos de ordenação		
OrderBy	OrderByDescending	ThenBy
ThenByDescending	Reverse	

Métodos de Agregação		
Average	Count	LongCount
Max	Min	Sum

Métodos de paginação		
First	FirstOrDefault	Last
LastOrDefault	Single	SingleOrDefault
Skip	SkipWhile	Take
TakeWhile		

Métodos de junção	
Join	GroupJoin

# Reflection



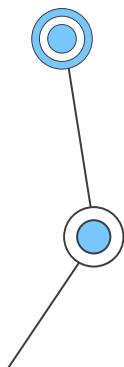
Antes de falar de **Reflection**, devemos entender o que é a **Metaprogramação**, pois o uso de reflection também é uma estratégia chave para a metaprogramação.

**Metaprogramação** é uma **técnica** de programação na qual os programas de computador têm a capacidade de tratar **outros programas** como se fosse **seus dados**. Isso significa que um programa pode ser projetado para ler, gerar, analisar ou transformar outros programas e até mesmo modificar a si mesmo em **tempo de execução**.

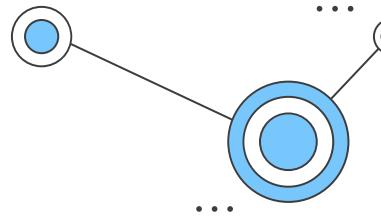
Em alguns casos, isso permite que os programadores minimizem o número de linhas de código para expressar uma solução, **reduzindo o tempo de desenvolvimento**. Ele também permite aos programas **maior flexibilidade** para lidar com novas situações com eficiência sem recompilação.

A capacidade de uma linguagem de programação ser sua própria metalinguagem é chamada de **Reflection** ou **Reflexão**. A reflexão é um recurso valioso da linguagem para facilitar a metaprogramação.

Reflection e metaprogramação são ferramentas poderosas, e o C# possui muitos metadados e com isso a presença deste mecanismo faz com que se pague um **custo muito alto** no **uso de memória ram**, por isso é necessário ter um bom **domínio da programação** para utilizar esse recurso da melhor forma, Então como já dizia o Tio Ben, com **grandes poderes** vem **grandes responsabilidades**.



# Reflection



O conceito de **Reflection** ou **Reflexão** é a capacidade de manipular qualquer **estrutura de dados** através dos **metadados** em **tempo de execução**, com isso é possível usar a reflexão para criar dinamicamente uma instância de um tipo, associar o tipo a um objeto existente ou obter o tipo de um objeto existente, **invocar seus métodos** ou **acessar suas propriedades e campos**, mesmo que você **não conheça** a estrutura de dados que queira manipular.

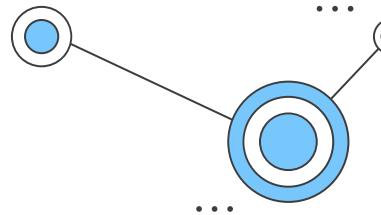
Tarefas que podemos realizar com reflection:

- Obter metadados das propriedades e métodos;
- Instanciar objetos, sem utilizar o operador new;
- Chamar métodos e alterar propriedades;
- Compilar novos tipos de dados em tempo de execução;
- Executar serialização ou desserialização de dados;
- Mapear propriedades de entidades com tabelas do banco de dados, isso é muito utilizado em ORMs;

Dessa forma podemos usar Reflection para realizar uma **programação genérica** onde podemos generalizar o código onde os nomes das propriedades e métodos podem variar.



# Reflection



Em poucas palavras, **Reflection** é a **capacidade de manipular** qualquer **estrutura de dados** através dos **metadados** e utilizá-lo como se fosse **seus dados** e tudo isso pode ser feito em **tempo de execução**.

Sabendo disso, vamos para exemplos de código que ficará mais claro o uso desta técnica.

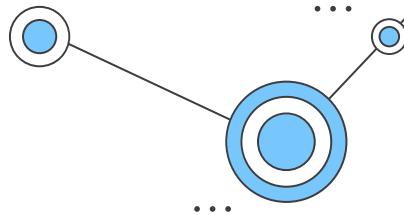
Vamos imaginar um cenário onde precisamos logar as informações das nossas classes de domínio em um projeto.

Temos as classes **Cliente**, **Produto** e **Pedido** e precisamos logar as informações.

Vou mostrar como podemos fazer isso **sem usar Reflection** e depois **com o uso de Reflection** que vai facilitar muito a nossa vida.



# Reflection



- ❑ A seguir temos o código de cada classe :

```
public class Pedido
{
    0 references
    public int Id { get; set; }
    0 references
    public int ClienteId { get; set; }
    0 references
    public DateTime DataPedido { get; set; }

}
```

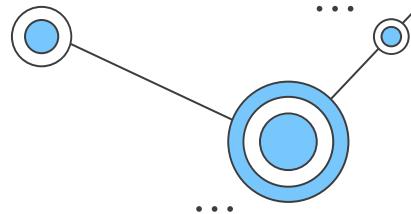
```
public class Cliente
{
    1 reference
    public int Id { get; set; }
    1 reference
    public string Nome { get; set; }
    1 reference
    public string Endereco { get; set; }

}
```

```
public class Produto
{
    0 references
    public int Id { get; set; }
    0 references
    public string Nome { get; set; }
    0 references
    public string Descricao { get; set; }
    0 references
    public Decimal Preco { get; set; }
    0 references
    public int Estoque { get; set; }

}
```

# Reflection



## □ Criando a classe de Log sem usar Reflection

Agora vamos fazer uma implementação bem simples da classe que vai logar as informações que precisamos, esta classe é bem simples e possui apenas os métodos para gerar e imprimir o log.

```
public static void LogProdutos(Produto produto)
{
    StringBuilder builder = new StringBuilder();
    builder.AppendLine("Log do produto");
    builder.AppendLine("Data: " + DateTime.Now);
    builder.AppendLine("Id: " + produto.Id);
    builder.AppendLine("Nome: " + produto.Nome);
    builder.AppendLine("Descrição: " + produto.Descricao);
    builder.AppendLine("Estoque: " + produto.Estoque);
    ImprimeLog(builder.ToString());
    //SalvaLog()
}
```

```
public static void LogPedidos(Pedido pedido)
{
    StringBuilder builder = new StringBuilder();
    builder.AppendLine("Log do pedido");
    builder.AppendLine("Data: " + DateTime.Now);
    builder.AppendLine("Id: " + pedido.Id);
    builder.AppendLine("ClienteId: " + pedido.ClienteId);
    builder.AppendLine("DataPedido: " + pedido.DataPedido);
    ImprimeLog(builder.ToString());
}
```

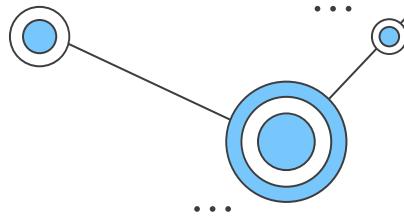
# Reflection

```
public static void LogClientes(Cliente cliente)
{
    StringBuilder builder = new StringBuilder();
    builder.AppendLine("Log do cliente");
    builder.AppendLine("Data: " + DateTime.Now);
    builder.AppendLine("Id: " + cliente.Id);
    builder.AppendLine("Nome: " + cliente.Nome);
    builder.AppendLine("Endereço: " + cliente.Endereco);
    ImprimeLog(builder.ToString());
}
```

```
public static void ImprimeLog(string texto)
{
    Console.WriteLine(texto);
}

0 references
public void SalvaLog(string texto)
{
    //salva o log
}
```

# Reflection



## ❑ Criando a classe de Log usando Reflection

Agora vamos fazer uma implementação do log usando Reflection e melhorar o nosso código tornando-o mais robusto e aderente às boas práticas.

```
public static void Log(object obj)
{
    var tipo = obj.GetType();

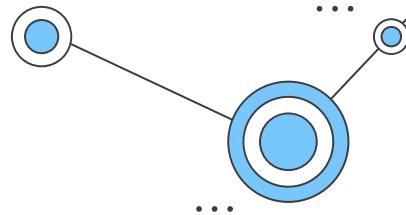
    StringBuilder builder = new StringBuilder();
    builder.AppendLine("Log do " + tipo.Name);
    builder.AppendLine("Data: " + DateTime.Now);

    foreach (var prop in tipo.GetProperties())
    {
        builder.AppendLine(prop.Name + ": " + prop.GetValue(obj));
    }

    ImprimeLog(builder.ToString());
}
```



# Reflection



## ❑ Vamos entender o funcionamento deste código:

Primeiro vamos criar um método genérico para criar um log para qualquer classe.

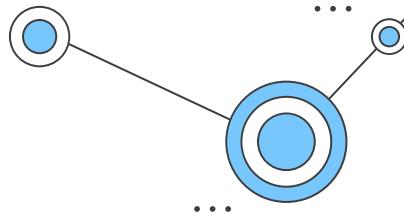
```
public static void Log(object obj)
{
}
```

o método **GetType()** obtém o nome do tipo esse tipo e não tem relação com a instância de obj

```
var tipo = obj.GetType();
```



# Reflection



com a propriedade **Name** conseguimos buscar o nome do tipo.

```
StringBuilder builder = new StringBuilder();

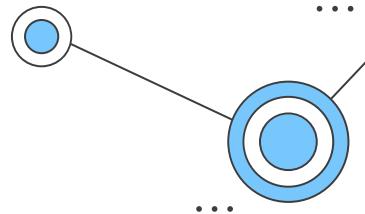
builder.AppendLine("Log do " + tipo.Name); ←
builder.AppendLine("Data: " + DateTime.Now);
```

Vamos obter agora todas as propriedades do tipo, usamos o método **GetProperties** para obter o nome das propriedades do tipo.

```
foreach (var prop in tipo.GetProperties())
{
```



# Reflection



Como já falamos antes, a propriedade **Name** é utilizada para obter o nome da propriedade e o método **GetValue()** para obter o valor da instância desse tipo.

```
foreach (var prop in tipo.GetProperties())
{
    builder.AppendLine(prop.Name + ":" + prop.GetValue(obj));
```

Método para imprimir o log no console.

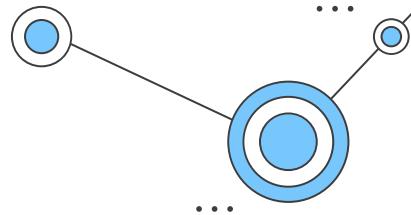
```
ImprimeLog(builder.ToString());
```

```
1 reference
public static void ImprimeLog(string texto)
{
    Console.WriteLine(texto);
}
```

Agora temos um único método genérico que pode ser usado para logar as informações de qualquer classe.



# Reflection



## ❑ Logando as informações

Apenas para mostrar que o código está funcional vamos implementar o código que cria instâncias das classes e usa as duas implementações do log.

Na Classe **Program.cs** no método **Main**, vamos criar nossos objetos e chamar os métodos **LogarSemReflection** e **LogarUsandoReflection**.

```
Console.WriteLine("***** Logando sem usar Reflection *****");
LogarSemReflection(cliente, produto, pedido);

Console.WriteLine(" ----- Logando usando Reflection -----");
LogarUsandoReflection(cliente, produto, pedido);
Console.ReadKey();
```

```
var cliente = new Cliente()
{
    Id = 10,
    Nome = "Willian Brito",
    Endereco = "Rua de teste, 0101"
};

var produto = new Produto()
{
    Id = 1,
    Nome = "Cubo Magico",
    Descricao = "Cubo Magico 3x3",
    Estoque = 100,
    Preco = 63.99M
};

var pedido = new Pedido()
{
    Id = 1,
    ClienteId = 1,
    DataPedido = DateTime.Now
};
```

# Reflection

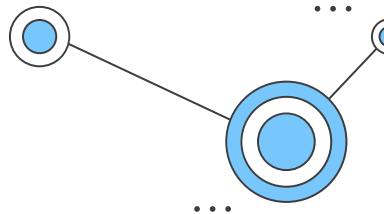
Implementação dos métodos **LogarSemReflection** e **LogarUsandoReflection**.

```
1 reference
public static void LogarSemReflection(Cliente cli, Produto prod, Pedido ped)
{
    LogSemReflection.LogClientes(cli);
    LogSemReflection.LogProdutos(prod);
    LogSemReflection.LogPedidos(ped);
}

1 reference
public static void LogarUsandoReflection(Cliente cli, Produto prod, Pedido ped)
{
    LogComReflection.Log(cli);
    LogComReflection.Log(prod);
    LogComReflection.Log(ped);
}
```

Repare que na implementação sem utilizar reflection utilizamos implementações especializadas de cada objeto, já no método utilizando reflection temos apenas um método que pode ser utilizado por qualquer tipo de estrutura, ou seja, utilizando reflection programamos de forma mais genérica, com menos escrita de código e com maior reutilização.

# Reflection



- Executando o projeto teremos o seguinte resultado :

```
***** Logando sem usar Reflection *****
Log do cliente
Data: 27/08/2022 10:43:55
Id: 10
Nome: Willian Brito
Endereço: Rua de teste, 0101

Log do produto
Data: 27/08/2022 10:43:55
Id: 1
Nome: Cubo Magico
Descrição: Cubo Magico 3x3
Estoque: 100

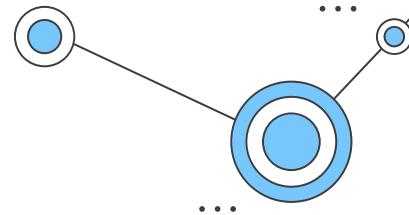
Log do pedido
Data: 27/08/2022 10:43:55
Id: 1
ClienteId: 1
DataPedido: 27/08/2022 10:43:55
```

```
----- Logando usando Reflection -----
Log do Cliente
Data: 27/08/2022 10:43:55
Id: 10
Nome: Willian Brito
Endereco: Rua de teste, 0101

Log do Produto
Data: 27/08/2022 10:43:55
Id: 1
Nome: Cubo Magico
Descricao: Cubo Magico 3x3
Preco: 63,99
Estoque: 100

Log do Pedido
Data: 27/08/2022 10:43:55
Id: 1
ClienteId: 1
DataPedido: 27/08/2022 10:43:55
```

# Reflection



Agora vamos aplicar Reflection para realizar a serialização de um objeto para o formato JSON.

```
public static string SerializarObjetoParaJson(object obj)
{
    var valorSerializado = "";

    foreach (var prop in obj.GetType().GetProperties())
    {
        var valor = prop.GetValue(obj);

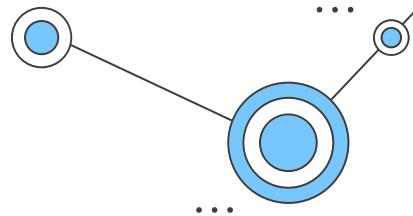
        if (prop.PropertyType == typeof(DateTime))
        {
            valorSerializado += "\"" + prop.Name + ":" + "\"" + String.Format("{0:dd/MM/yyyy}", valor) + ",";
        }
        else if (prop.PropertyType == typeof(Decimal) || prop.PropertyType == typeof(Double))
        {
            valorSerializado += "\"" + prop.Name + ":" + "\"" + String.Format("{0:N}", valor) + ",";
        }
        else
        {
            valorSerializado += "\"" + prop.Name + ":" + "\"" + valor + "\",";
        }
    }

    valorSerializado = valorSerializado.Substring(1, valorSerializado.Length - 2);

    return "{\"" + valorSerializado + "\"}";
}
```



# Reflection



- ❑ Vamos entender o funcionamento deste código:

Primeiro vamos criar um método genérico para serializar um objeto.

```
3 references
public static string SerializarObjetoParaJson(object obj)
{
}
```

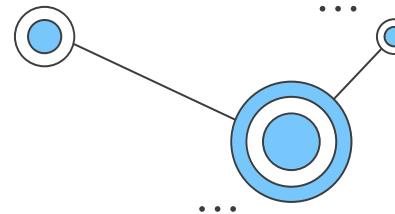
Vamos obter agora todas as propriedades do tipo usando o método **GetProperties** para obter o nome das propriedades do tipo.

```
var valorSerializado = "";

foreach (var prop in obj.GetType().GetProperties())
{
}
```



# Reflection



- ❑ Vamos entender o funcionamento deste código:

Usamos o método **GetValue** para obter o valor da instância desse tipo.

```
var valor = prop.GetValue(obj);
```

Agora verificamos o tipo de cada propriedade para realizar a formatação adequada.

```
if (prop.PropertyType == typeof(DateTime))
{
    valorSerializado += "\"" + prop.Name + ":" + "\"" + String.Format("{0:dd/MM/yyyy}", valor) + ",";
}
else if (prop.PropertyType == typeof(Decimal) || prop.PropertyType == typeof(Double))
{
    valorSerializado += "\"" + prop.Name + ":" + "\"" + String.Format("{0:N}", valor) + ",";
}
else
{
    valorSerializado += "\"" + prop.Name + ":" + "\"" + valor + ",";
}
```

# Reflection

Retirando a ultima virgula.

```
valorSerializado = valorSerializado.Substring(1, valorSerializado.Length - 2);
```

Retornando o objeto serializado.

```
return "{\"" + valorSerializado + "\"};
```

Utilizando a função **SerializarObjetoParaJson** no objeto **cliente**, **pedido** e **produto**.

```
var resultado1 = SerializarObjetoParaJson(cliente);
var resultado2 = SerializarObjetoParaJson(pedido);
var resultado3 = SerializarObjetoParaJson(produto);

Console.WriteLine(resultado1);
Console.WriteLine("");
Console.WriteLine(resultado2);
Console.WriteLine("");
Console.WriteLine(resultado3);
```

# Reflection

- ❑ Executando o projeto teremos o seguinte resultado :

```
~/Área de trabalho/App/Microsoft/Reflection/ExemploReflection ➔ dotnet run
{"Id":"10","Nome":"Willian Brito","Endereco":"Rua de teste, 0101"}
{"Id":"1","ClienteId":"1","DataPedido":"27/08/2022"}
{"Id":"1","Nome":"Cubo Magico","Descricao":"Cubo Magico 3x3","Preco":63,990,"Estoque":100}
```

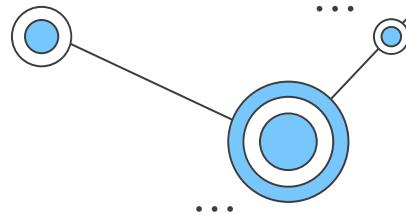
Podemos verificar se o resultado possui o formato JSON correto realizando a validação no site :  
<https://jsonformatter.curiousconcept.com/>

**VALID (RFC 8259)**

**Formatted JSON Data**

```
{
  "Id": "1",
  "Nome": "Cubo Magico",
  "Descricao": "Cubo Magico 3x3",
  "Preco": 63,990,
  "Estoque": 100
}
```

# Reflection



Usar reflexão não é só uma tarefa para descobrir tipos, existe a possibilidade de refletir métodos encapsulados em uma DLL.

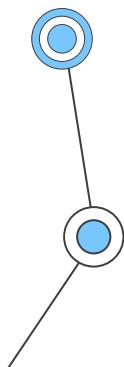
Antes de efetuar o load de uma DLL, a reflexão permite também a geração de novas instâncias de um tipo, utilizando a classe **Activator**. Isso é só um exemplo das muitas possibilidades do reflection.

Vamos refletir a classe **Produto** para que possamos exemplificar todos os principais pontos do Reflection.

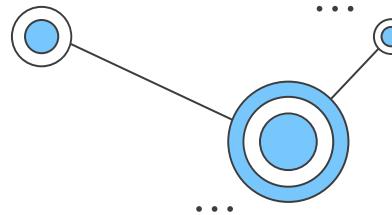
```
UtilizandoReflection(produto);
```

Para conhecer o assembly da classe Produto:

```
public static void UtilizandoReflection(object obj)
{
    var assembly = obj.GetType().Assembly;
    Console.WriteLine(assembly);
}
```



# Reflection



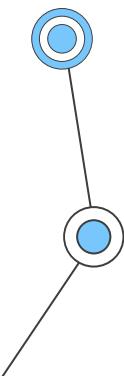
Veja que o seguinte valor irá aparecer no objeto, onde ExemploReflection é o nome do namespace que comporta a classe Produto, a versão 1.0 é a versão do assembly no momento da compilação.

```
ExemploReflection, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
```

Agora vou mostrar algumas coisas que conseguimos fazer utilizando reflection.

Esta linha equivale ao operador **New**.

```
var pessoa = Activator.CreateInstance(obj.GetType());
```



# Reflection

Vou criar uma classe pessoa para demonstrar como podemos acessar propriedades, alterar valores e invocar métodos da classe em tempo de execução utilizando reflection.

```
public class Pessoa
{
    2 references
    public string Nome { get; set; }
    2 references
    public int Idade { get; set; }
    2 references
    public decimal Altura { get; set; }
    2 references
    public DateTime DataNascimento { get; set; }

    0 references
    public void Caracteristicas()
    {
        Console.WriteLine($"Nome: {this.Nome}");
        Console.WriteLine($"Idade: {this.Idade}");
        Console.WriteLine($"Altura: {this.Altura}");
        Console.WriteLine($"DataNascimento: {this.DataNascimento}");
    }
}
```

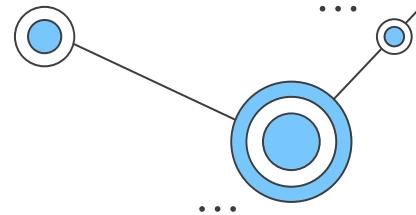
# Reflection

Vamos instanciar a classe que acabamos de criar.

```
var pessoa = new Pessoa()
{
    Nome = "Monara Isabela Poli",
    Idade = 28,
    Altura = 1.65M,
    DataNascimento = Convert.ToDateTime("03/12/1993")
};
```

Vou passar essa instância para o método de exemplo.

```
pessoa.BuscarCaracteristicas();
UtilizandoReflection(pessoa);
```



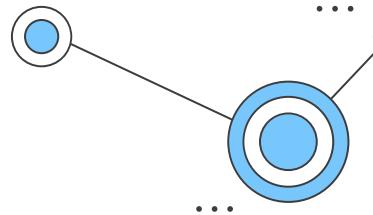
# Reflection

```
reference
public static void UtilizandoReflection(object obj)
{
    var pessoa = Activator.CreateInstance(obj.GetType());
    var listaPropriedades = pessoa.GetType().GetProperties();

    foreach (var prop in listaPropriedades)
    {
        if (prop.PropertyType == typeof(DateTime))
        {
            prop.SetValue(pessoa, Convert.ToDateTime("01/05/1991"));
        }
        else if (prop.PropertyType == typeof(Decimal) || prop.PropertyType == typeof(Double))
        {
            prop.SetValue(pessoa, 1.80M);
        }
        else if (prop.PropertyType == typeof(Int32) || prop.PropertyType == typeof(Int64))
        {
            prop.SetValue(pessoa, 31);
        }
        else
        {
            prop.SetValue(pessoa, "Willian Ferreira Brito");
        }
    }

    pessoa.GetType().InvokeMember("BuscarCaracteristicas", BindingFlags.InvokeMethod | BindingFlags.Public | BindingFlags.Instance, null, pessoa, null);
}
```

# Reflection



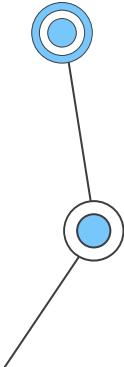
No último exemplo o código estava acessando as propriedades da classe pessoa em tempo de execução alterando os valores depois invocando o método **Características**.

**Executando o projeto teremos o seguinte resultado :**

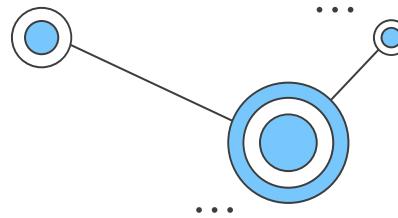
```
Nome: Monara Isabela Poli
Idade: 28
Altura: 1,65
DataNascimento: 03/12/1993 00:00:00

Nome: Willian Ferreira Brito
Idade: 31
Altura: 1,80
DataNascimento: 01/05/1991 00:00:00
```

Vale a pena ressaltar que métodos privados não são visualizados de imediato pela reflexão, entretanto é possível executá-los utilizando os **BindingFlags** apropriados para isso.



# Reflection

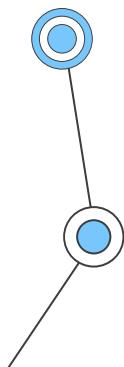


## ❑ BindingFlags

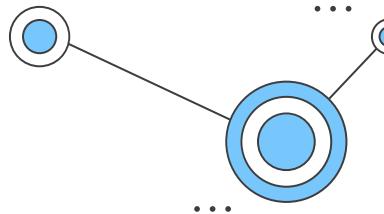
É um enumerador com diversas opções que servem como parâmetros para o momento da reflexão. E para utilizar reflection da melhor forma é importante conhecê-lo para usufruir melhor da reflexão.

### **Link da documentação:**

<https://docs.microsoft.com/en-us/dotnet/api/system.reflection.bindingflags?redirectedfrom=MSDN&view=net-6.0>



# Reflection

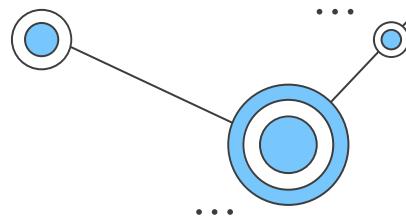


Vale ressaltar que é possível utilizar métodos que possuem parâmetros de entrada, vamos criar um método com parâmetros de entrada e mostrar um exemplo de como ficaria utilizando reflection neste caso:

```
    public void BuscarCaracteristicasComParametro(Pessoa pessoa)
    {
        Console.WriteLine($"Nome: {pessoa.Nome}");
        Console.WriteLine($"Idade: {pessoa.Idade}");
        Console.WriteLine($"Altura: {pessoa.Altura}");
        Console.WriteLine($"DataNascimento: {pessoa.DataNascimento}");
    }
}
```

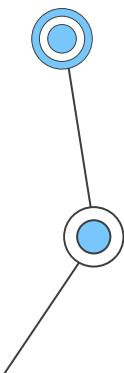
```
pessoa.GetType().InvokeMember("BuscarCaracteristicasComParametro",
    BindingFlags.InvokeMethod | BindingFlags.Public | BindingFlags.Instance,
    null,
    pessoa,
    new object[] { pessoa } );
```

# Reflection

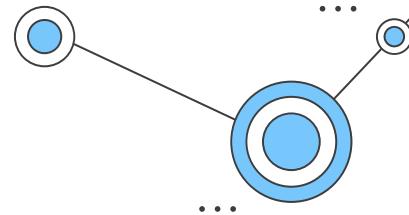


## ❑ Concluindo

O Reflection ajuda os programadores a criar bibliotecas de software genéricas para exibir dados, processar diferentes formatos de dados, executar serialização ou desserialização entre outras coisas, ou seja, tudo que tem por de baixos dos panos de compiladores e grandes frameworks tem o uso de reflection, neste conteúdo vimos apenas uma ponta do iceberg, então vale a pena se aprofundar neste assunto, caso queira criar funcionalidades mais robustas e com maior reutilização.



# Processamento Assíncrono

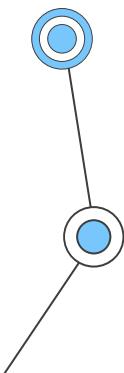


Programação assíncrona permite que a sua aplicação responda de maneira mais rápida e com fluidez, mas pode também causar bastante problema e confusão se não for utilizada da maneira correta.

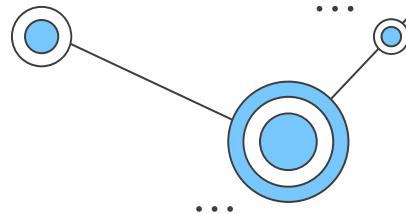
## □ O que é Programação Assíncrona

Vamos entender alguns conceitos sobre os tipos de programação:

- Síncrono
- Assíncrono
- MultiThreading
- Concorrente
- Paralelismo



# Processamento Assíncrono



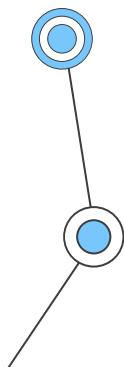
**Síncrono:** Quando você executa um código de maneira síncrona, significa que é executado na ordem em que foi escrito e você precisa esperar a etapa do código anterior terminar para poder prosseguir.

**Assíncrono:** Neste modelo você **não precisa esperar** a etapa do código anterior terminar para poder prosseguir. É comum neste modelo de programação o uso de “call-backs”, que são chamados ao final da operação.

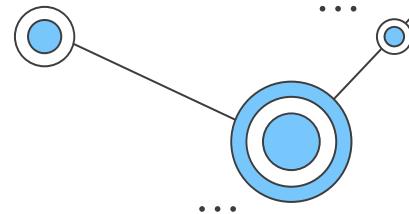
**MultiThreading:** No multithreading você pode executar vários códigos ao mesmo tempo. Entenda que uma “thread” é uma execução de código e podemos ter dezenas, centenas delas executando neste exato momento em seu computador. O Windows é um sistema multithread, pois consegue realizar várias tarefas em paralelo, como copiar um arquivo enquanto você navega na internet. Só é possível multithreading com paralelismo.

**Concorrência:** É sempre que você tiver múltiplos processos ou threads manipulando os mesmos dados ou **disputando os mesmos recursos**.

**Paralelismo:** Neste cenário temos duas ou mais tarefas sendo executadas **literalmente ao mesmo tempo**, por exemplo em um computador com vários núcleos (cores).



# Processamento Assíncrono

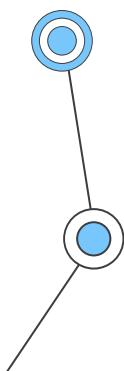


## □ Diferença entre Concorrência e Paralelismo

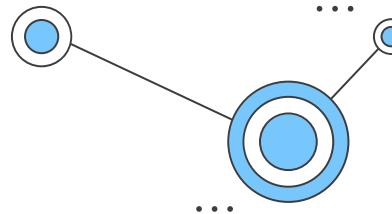
Concorrência é basicamente a capacidade de lidar com várias coisas de uma só vez, enquanto o paralelismo é a capacidade de lidar com várias coisas ao mesmo tempo.

Se você achou que parece ser a mesma coisa, calma... vamos dar um exemplo para tentar deixar mais claro.

Imagine que existem dois componentes independentes em seu navegador, um responsável por gerenciar download de arquivos e outro pela renderização do programa. Em um programa que utiliza concorrência, quando estivermos efetuando o download de um arquivo, o programa vai ficar alterando a execução desses dois componentes, para que consiga baixar um pedaço do arquivo e na sequência renderizar essa informação no navegador.



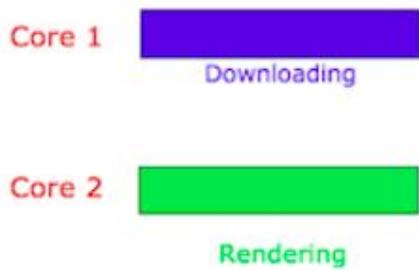
# Processamento Assíncrono



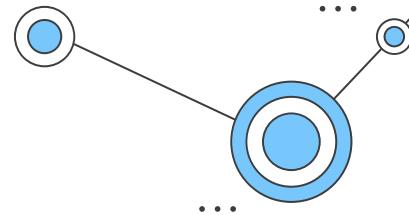
Agora imagine o mesmo navegador funcionando em um computador multi-core, onde ele consegue fazer download e renderizar a informação ao mesmo tempo. Nesse caso o programa estaria utilizando paralelismo.

Embora pelas imagens utilizar paralelismo pareça ser a melhor opção, nem sempre é, pois os dois componentes precisam “conversar” entre si, para que o componente de renderização saiba o percentual ou se o download já acabou.

O custo dessa “conversa” em sistemas funcionando com concorrência é muito baixo. Já em sistemas rodando com paralelismo, esse custo é muito alto, o que pode levar o sistema a ser mais lento.



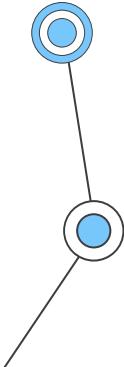
# Processamento Assíncrono



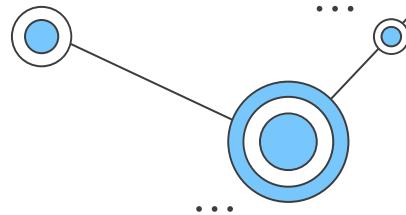
## Quando devemos usar síncrono ou assíncrono

Se você tem processamentos longos, como acesso a dados, rede, leitura e escrita de arquivos, você pode usar programação assíncrona. Códigos com alto uso de processamento também são candidatos a programação assíncrona. Mas se o seu código tem um fluxo contínuo e cada operação precisa sempre aguardar a anterior, você pode usar o modelo síncrono. Vale a pena considerar tornar o código assíncrono sempre que possível, pois isto aumenta a performance.

**Um alerta importante:** a programação assíncrona usa mais recursos do computador, então é importante dimensionar corretamente a máquina, pois você terá muito mais código sendo executado ao mesmo tempo, e isto também pode aumentar a complexidade na hora de fazer depuração (debug).



# Processamento Assíncrono



## □ Operações I/O Bound e CPU Bound

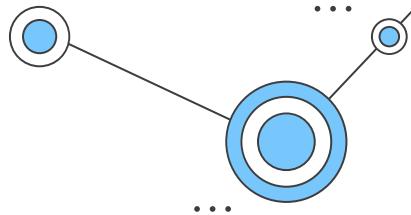
Quando falamos de processamento assíncrono, temos que levarem consideração dois aspectos:

**I/O Bound:** operações que acessam recursos como disco, rede para realizar uma operação, por exemplo:

Neste exemplo, estamos baixando o conteúdo do site da Microsoft, como um texto, então estamos usando recursos de rede para acessar o site. A palavra “**async**” é quem permite a execução assíncrona e iremos falar dela mais adiante.

```
var web = new HttpClient();
var site = await web.GetStringAsync(new Uri("https://www.microsoft.com"));
```

# Processamento Assíncrono



**CPU bound:** operações que executam muitos cálculos, que utilizam muito o processador da máquina, por exemplo:

Neste exemplo, estamos ordenando uma lista com **Sort()**, então estamos usando processamento.

## Async / Await

Para programarmos de maneira assíncrona vamos utilizar sempre dois objetos: **Task** e **Task<T>**, que permitem este tipo de operação. Também utilizaremos as palavras chave **async** e **await**, que é onde tudo acontece. Estas duas palavras-chave são o ponto central da programação assíncrona.

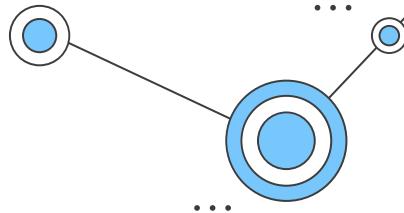
A palavra **await** inicia a execução assíncrona do código, controlando o fluxo de execução e retornando ao ponto da chamada para continuar o fluxo. E a palavra **async** permite o código ser executado de maneira assíncrona.



```
var lista = new List<string>();

lista.Add("Maria");
lista.Add("Joao");
lista.Add("Antonio");
lista.Add("Joaquim");
lista.Sort();
```

# Processamento Assíncrono



Para entendermos melhor como funciona o fluxo de execução do **async/await**, veja o exemplo a seguir.

Aqui temos a classe **Program** com método **Main**, neste programa o usuário deve informar o ID do cliente e o método **GetCustomerOrders** retornará todos os pedidos do cliente que foi informado pelo usuário e depois exibe a mensagem de **aguardando**.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Informe o ID do cliente");

        var criterio = Console.ReadLine().ToUpper();

        OrderService.GetCustomerOrders(criterio);

        Console.WriteLine("Aguardando...");
        Console.ReadLine();
    }
}
```



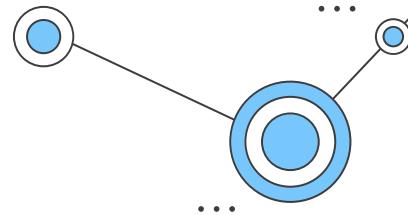
# Processamento Assíncrono

O método **GetCustomerOrders** recebe uma string com nome criterio que equivale ao ID do cliente o método conecta no banco retorna uma lista de pedidos do cliente informado.

```
1 reference
public class OrderService
{
    1 reference
    public static void GetCustomerOrders(string criterio)
    {
        try
        {
            using (var db = new AppDbContext())
            {
                var pedidos = db.Orders
                    .Where(c => c.CustomerID.Contains(criterio))
                    .ToList();

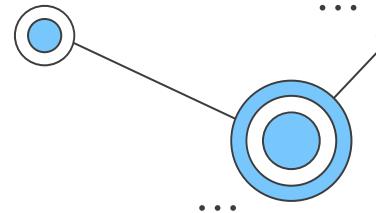
                ExibePedidos(pedidos);
            }
        }
        catch (Exception)
        {
            throw;
        }
    }
}
```

# Processamento Assíncrono

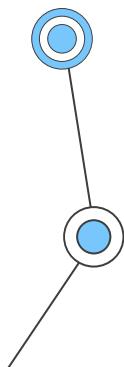


O método **ExibePedidos** recebe como entrada uma lista de pedidos e imprime na tela todos os pedidos desta lista.

# Processamento Assíncrono



No modelo **síncrono** o método **Main** é iniciado na **thread principal** do programa, quando é chamado o método **GetCustomerOrders**, a thread principal é **bloqueada**, o método GetCustomerOrders faz a consulta no banco de dados e só depois que for retornado os pedidos deste cliente é chamado o método **ExibePedidos**, com isso a thread principal é desbloqueada e só assim ele exibe a mensagem de aguardando.



# Processamento Assíncrono

Repare na tela do usuário que a mensagem de **aguardando** só é exibida depois do resultado do método **GetCustomersOrders**.

```
Informe o ID do cliente
vinet
Pedidos do cliente : VINET
ID : 10248 - Data 04/07/1996 00:00:00 - Destino Reims
ID : 10274 - Data 06/08/1996 00:00:00 - Destino Reims
ID : 10295 - Data 02/09/1996 00:00:00 - Destino Reims
ID : 10737 - Data 11/11/1997 00:00:00 - Destino Reims
ID : 10739 - Data 12/11/1997 00:00:00 - Destino Reims
Aguardando... ←
```

# Processamento Assíncrono

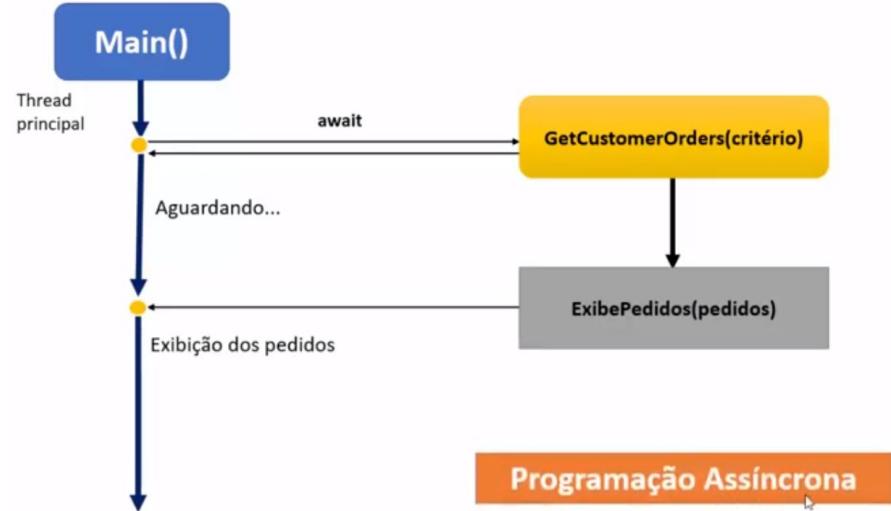
Agora vamos refatorar método **GetCustomersOrders** para trabalhar de forma assíncrona, repare que acrescentamos o modificador **async** para falarmos para o C# que este método será assíncrono, depois substituímos o método de **ToList** para **ToListAsync** que retorna uma tarefa com uma lista de pedidos (Task<List<Order>>) e para controlar o fluxo do programa acrescentamos a palavra-chave **await** para **esperar o resultado** do método **ToListAsync**.

```
1 reference
public async static void GetCustomerOrders(string criterio)
{
    try
    {
        using (var db = new AppDbContext())
        {
            var pedidos = await db.Orders ←
                I.Where(c => c.CustomerID.Contains(criterio))
                .ToListAsync(); ←

            ExibePedidos(pedidos);
        }
    }
    catch (Exception)
    {
        throw;
    }
}
```

# Processamento Assíncrono

No modelo **assíncrono** o método **Main** é iniciado na **thread principal** do programa e quando é chamado o método **GetCustomerOrders** que agora é assíncrono e possui a palavra-chave **await**, irá **aguardar** o resultado da tarefa (Task) e o fluxo do programa retornará para a thread principal, logo após o retorno é exibido a mensagem **aguardando** e depois quando for retornado os pedidos do cliente do método **GetCustomerOrders** é chamado o método **ExibePedidos** que imprime todos os pedidos na tela do usuário.

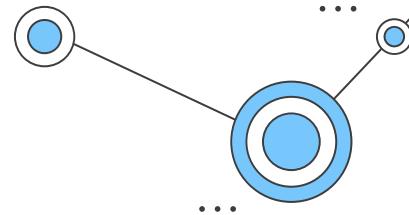


# Processamento Assíncrono

Repare na tela do usuário que a mensagem de **aguardando** agora é exibida antes do resultado do método **GetCustomersOrders**, isso acontece porque a thread principal não é bloqueada e o fluxo do programa continua graças a **programação assíncrona**.

```
Informe o ID do cliente
vinet
Aguardando... ←
Pedidos do cliente : VINET
ID : 10248 - Data 04/07/1996 00:00:00 - Destino Reims
ID : 10274 - Data 06/08/1996 00:00:00 - Destino Reims
ID : 10295 - Data 02/09/1996 00:00:00 - Destino Reims
ID : 10737 - Data 11/11/1997 00:00:00 - Destino Reims
ID : 10739 - Data 12/11/1997 00:00:00 - Destino Reims
```

# Processamento Assíncrono

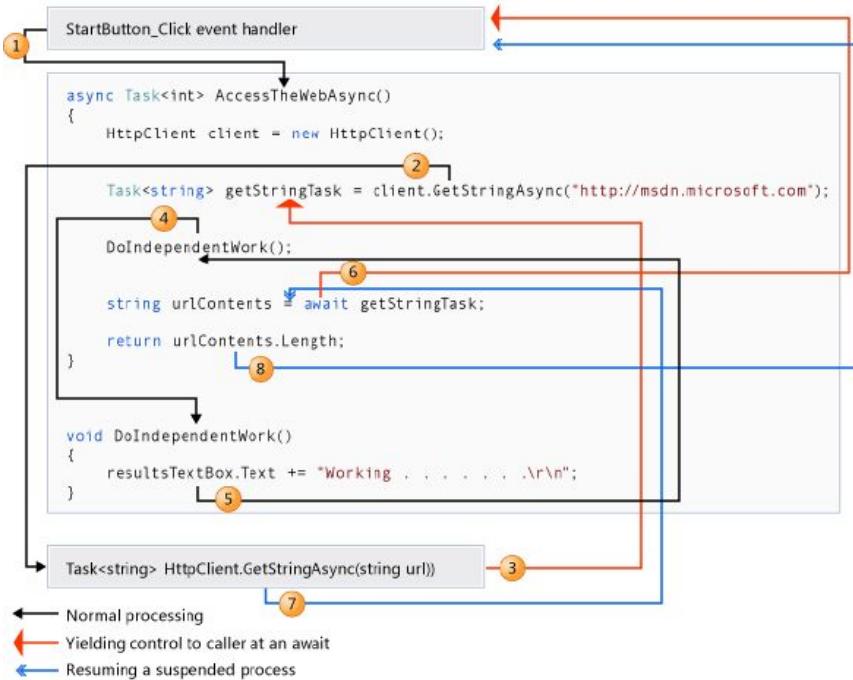


- Agora vamos falar sobre pontos obrigatórios para que a programação assíncrona aconteça no C#.
  - Você usa a palavra **async** na declaração de um método que dependa da palavra-chave **await**.
  - Ao usar o **await**, seu programa deve **esperar** um resultado (**Task**).
  - A palavra-chave **await** espera até que o método **retorne o resultado** sem bloquear o fluxo principal.
  - Vale a pena ressaltar que todo código que será executado pelo **await** deve **obrigatoriamente** retornar uma Task ou um objeto derivado de Task.
  - Um **await** só pode ser usado em um método declarado com o modificador **async**.
  - A Classe **Task** do **C#**, é bem semelhante ao conceito de **Promisse** da linguagem **Javascript**, que tem o conceito de algo que **será resolvido** no **futuro**.

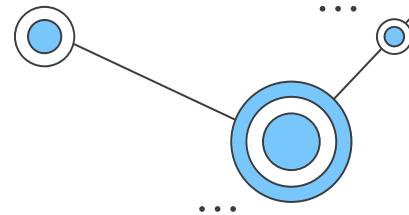


# Processamento Assíncrono

Agora vamos para outro exemplo de como funciona o fluxo de execução do **async/await**, vejamos a Figura a seguir.



# Processamento Assíncrono

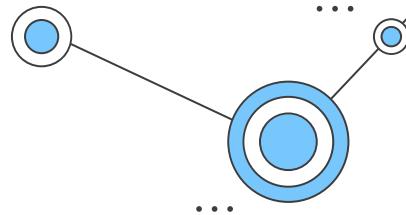


## □ Vamos entender o fluxo da aplicação:

1. Um evento de botão é disparado e chama um método assíncrono.
1. O método **AccessTheWebAsync()** cria um objeto **HttpClient** e chama **GetStringAsync()**.
1. O método **GetStringAsync()** inicia a execução, mas como ele é assíncrono, o fluxo é retornado para o chamador (`Access-TheWebAsync`). Ele deve retornar uma string **getStringTask**, então o processo segue para onde a string não é necessária.
1. Como o método **DoIndependentWork** não precisa do **getStringTask**, ele pode ser executado.
1. O método **DoIndependentWork()** realiza seu trabalho e retorna.
1. Para preencher a variável **urlContents** é preciso o valor de **getStringTask**, então o fluxo é novamente retornado para o chamador até que o valor seja retornado.
1. **GetStringASync()** processa os dados e retorna então, o controle para o ponto onde o valor era necessário.
1. Finalmente o tamanho da **string** é calculado e retornado.



# Processamento Assíncrono

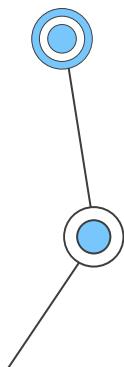


## Conclusão

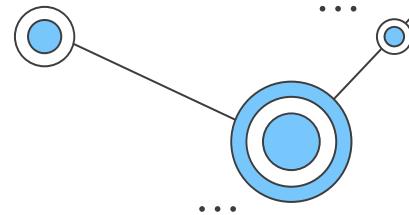
A programação assíncrona pode ser aplicada a diversos tipos de Aplicação, principalmente em aplicação web, lembrando que só não é recomendável utilizar essa técnica quando o código tem um fluxo contínuo e cada operação precisa sempre aguardar a anterior, porém vale a pena considerar tornar o código assíncrono sempre que possível.

Agora falando em boas práticas em operações **CPU Bound**, onde temos uso apenas de CPU, podemos criar métodos assíncronos com o método **Task.Run()**. Mas fique atento para usar este método somente em códigos que fazem uso de CPU e não uso de I/O.

Outra boa prática é sempre que você criar um método assíncrono, coloque a extensão **Async** no final do nome Ex: **“GetStringAsync()”**, pois isto indicará que o método aceita ser executado com **await**.



# Threads



## □ Porque Utilizar Threads

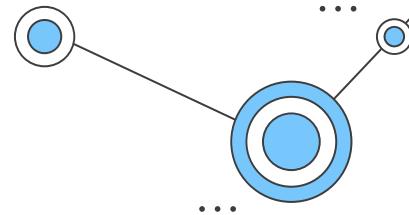
As threads são unidades mínimas de processamento usadas pelo sistema operacional para executar o código de programas e sistemas pelo processador. Ela existe dentro de um processo, isso quer dizer que um processo pode ter uma ou mais threads. Um processo sem threads não executaria código algum.

Elas também permitem que códigos sejam executados em paralelo e, com isso, agilizem tarefas de um programa ou Sistema Operacional. Por isso, é possível escutarmos música enquanto falamos no chat e ainda navegamos na internet em nossos computadores.

Com a chegada dos processadores com mais de um núcleo, a execução das threads em paralelo ficou muito mais eficiente. Uma das grandes vantagens em utilizá-las no desenvolvimento de sistemas é a paralelização de tarefas no seu código, gerando o ganho de tempo na execução.



# Threads



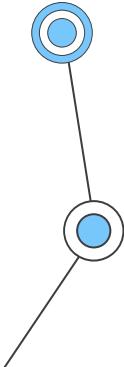
Os conceitos citados até agora são independentes da tecnologia usada no desenvolvimento de programas e sistemas, como por exemplo, C/C++, .NET (C#, VB.NET), Java etc.

A partir de agora, vamos entender as duas principais formas de trabalhar com threads no .NET Framework, suas facilidades, vantagens e desvantagens. Basicamente, o .NET Framework oferece duas maneiras de utilizar threads. Uma é através da classe `System.Threading.Thread`, e a outra através do .NET Thread Pool.

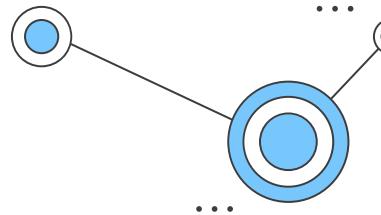
Vamos ao primeiro exemplo utilizando a classe `System.Threading.Thread`.

```
static void Main(string[] args)
{
    System.Threading.Thread threadAux = new System.Threading.Thread(ExutarThreadaux);
    threadAux.Start();
}

static void ExutarThreadaux()
{
    // Código executado pela thread aux
}
```



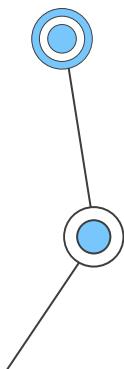
# Threads



Quando executamos esse código, uma nova thread é criada, o método **ExecutarThreadaux()** é executado e, posteriormente, a thread é finalizada e retirada da memória. A cada execução deste código, os passos são repetidos.

Porém, como a criação de threads é considerada um processo custoso (por envolver chamadas ao sistema operacional **[system call]** ), este método é indicado somente para quando o código a ser executado pela thread tiver uma duração longa e não necessitar recriar a thread a todo momento.

Para os casos de execução de código mais curtos, o .NET Framework disponibiliza o **ThreadPool** . Durante a inicialização de qualquer processo desenvolvido em .NET, um pool de threads é criado e usado pelo próprio .NET para diversas execuções do framework. Este pool também está disponível e pode ser utilizado por qualquer desenvolvedor na plataforma .NET, basta usar o objeto estático `ThreadPool` .



# Threads

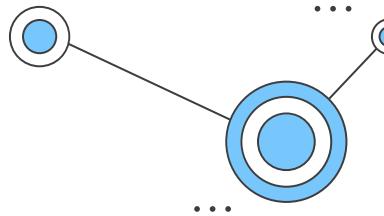
Vamos a outro exemplo de utilização de threads no .NET, mas agora utilizando o .NET ThreadPool.

Executando o código desse exemplo, o método **ExcutarThreadAux()** será colocado em uma fila e será executado pela próxima thread disponível no pool. Neste ponto, você pode estar pensando que colocar o método a ser executado em uma fila faça-o demorar mais a ser executado comparado a criação de thread pelo System.Threading.Thread . Porém isso não é verdade, pois o .NET ThreadPool foi criado pensando na reutilização de threads.

```
static void Main(string[] args)
{
    ThreadPool.QueueUserWorkItem(new WaitCallback(ExcutarThreadaux));
}

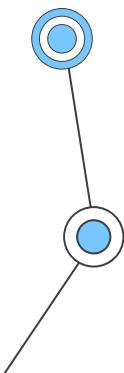
static void ExcutarThreadAux(object state)
{
    // Código executado pela threadaux
}
```

# Threads

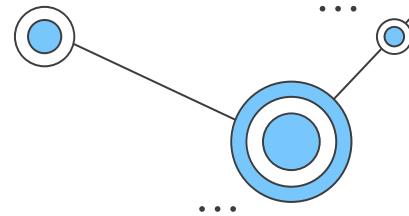


Assim, elas são criadas, utilizadas e, em vez de serem retiradas da memória quando acabam de executar, voltam ao ThreadPool , ficando disponíveis para um novo trabalho e evitando o custo futuro de criação de uma nova thread.

Além disso, o ThreadPool cria e executa as threads de uma forma controlada, levando em consideração a capacidade de hardware do computador (nº de processadores). Um exemplo simples de controle é quando o computador está com um consumo maior que 80% de CPU, e for solicitada a execução de um código pelo ThreadPool , conforme o exemplo no anterior. Em vez de disparar mais uma thread para concorrer por CPU e piorar o consumo de recursos da máquina, ele vai aguardar até o CPU cair abaixo de 80% para disparar os trabalhos na fila para as threads.



# Threads



## □ **System.Threading.Thread VS ThreadPool**

O .NET ThreadPool é recomendado na maioria dos casos de utilização de threads no .NET, por fazer diversos controles e garantir a performance da máquina onde o código é executado.

Porém, há situações onde o desenvolvedor precisa de um maior controle sobre a execução da thread e seu tempo de vida. Para esses casos de maior flexibilidade e controle, temos o `System.Threading.Thread`.



# Threads VS Task

**Threads:** Linha de execução (**em inglês: Thread**), é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente. (**System.Threading**).

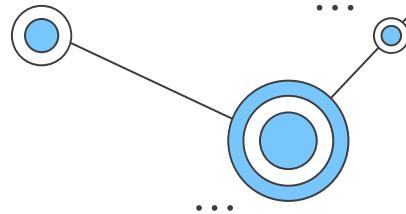
**Tasks:** Uma tarefa (ou task) representa uma unidade de trabalho que deverá ser realizada. (**System.Threading.Tasks**)

O exemplo de código a seguir nos mostra como podemos criar uma tarefa com a classe **Thread** e **Task** em C#.

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace Cshp_TaskThreds
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Iniciando uma Thread com delay de 5 s");
            //Threads
            Thread th = new Thread(() =>
            {
                Thread.Sleep(5000);
            });

            Console.WriteLine("Iniciando uma Task com delay de 5 s");
            //Tasks
            Task t = Task.Factory.StartNew(() =>
            {
                Thread.Sleep(5000);
            });
            Console.ReadLine();
        }
    }
}
```

# Threads VS Task

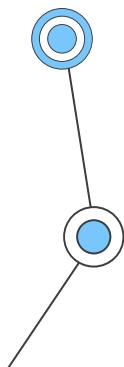


## □ Thread

A classe Thread cria um thread real no **nível do sistema operacional**. O encadeamento criado com a classe Thread consome recursos como memória para a pilha e a sobrecarga da CPU para as mudanças de contexto de um encadeamento para outro. A classe Thread fornece o mais alto grau de controle, como as funções Abort(), Suspend(), Resume() etc. Também podemos especificar alguns atributos de nível de thread, como tamanho da pilha .

## □ Task

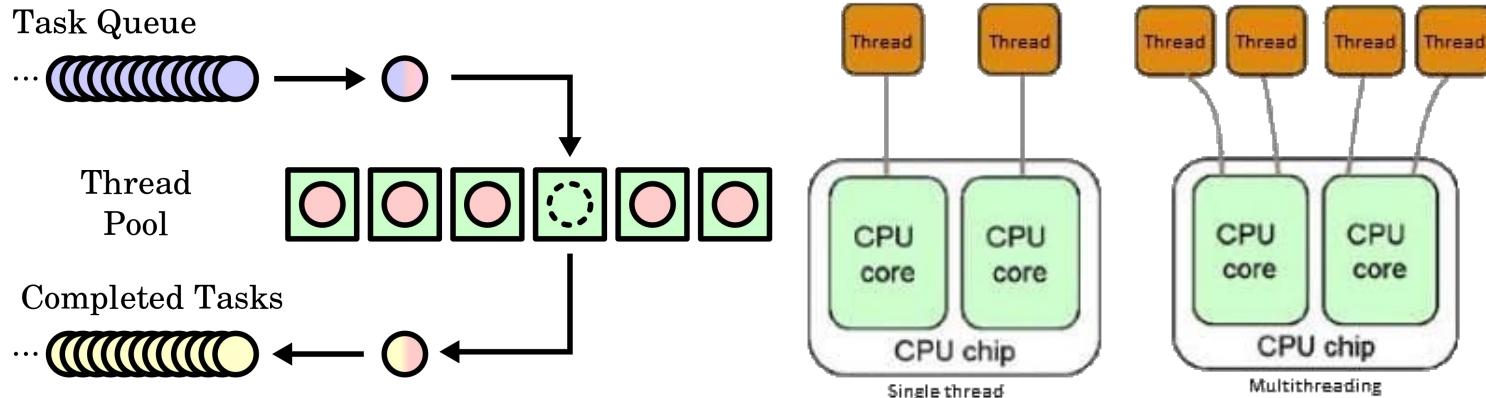
Uma classe Task cria uma tarefa assíncrona no **nível do software**. É algo que foi criado no .NET para o programador não ter que lidar com os detalhes do paralelismo ou assincronicidade. O agendador de tarefas executa as tarefas criadas com a classe Task. O agendador padrão executa a tarefa dentro do pool de threads. Ao contrário dos threads criados com a classe Thread, as tarefas criadas com a classe Task não requerem memória adicional ou recursos de CPU. A classe Task não pode ser usada para especificar os atributos de nível de thread, como o tamanho da pilha. Uma vez que a classe Task é executada no pool de threads, qualquer tarefa de longa execução pode preencher o pool de threads e a nova tarefa pode acabar esperando que as tarefas anteriores concluam a execução.



# Threads VS Task

Na plataforma .NET, uma **Task** representa uma operação assíncrona. Já as **Thread(s)** são usadas para concluir essa operação, quebrando o trabalho em pedaços e atribuindo-os as **threads(segmentos)** separados. Assim, uma **Task (Tarefa)** é uma promessa, ou seja, uma **Task<T>** promete devolver um T, e isso pode não ser agora, pode ser mais tarde.

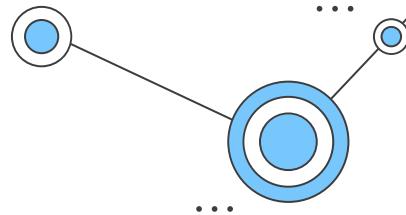
Uma **Thread** é uma forma de cumprir essa promessa. Mas nem todas as Tasks precisam de uma nova Thread. Na verdade, criar uma thread quase sempre não é desejável, pois tem um alto custo, assim é mais fácil reutilizar uma thread existente do threadpool.



# Threads VS Task

- Dessa forma uma Task possui as seguintes vantagens em termos de recursos sobre as threads:**
  - Se o sistema tiver várias Tasks, então ele faz uso do pool de threads interno da CLR e, portanto, não tem a sobrecarga associada à criação de uma thread dedicada usando uma Thread.
  - Uma Task pode retornar um resultado. Não há mecanismo direto para retornar o resultado de uma Thread.
  - Podemos encadear Tasks em conjunto para executar uma após a outra.
  - Estabelece um relacionamento pai / filho quando uma tarefa é iniciada a partir de outra tarefa. Uma exceção de uma Task filha pode propagar para a Task pai.
  - Ao usar uma thread, em casos de ocorrer uma exceção não é possível capturá-la na função Pai.
  - Uma Task suporta o cancelamento de tarefas através do uso de tokens de cancelamento.
  - Podemos usar as palavras chaves '**async**' e '**await**' para implementar facilmente operações assíncronas em uma Task.

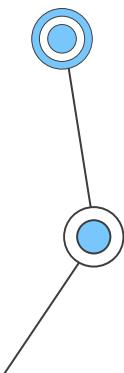
# Processamento Assíncrono



## Conclusão

Uma **Thread** é um fluxo separado de execução dentro de um programa, então quando temos um programa ele pode executar varias threads ao mesmo tempo, o que significa que vários fluxos de código estão sendo executados simultaneamente e a criação de uma thread é muito custosa, pois é executada em nível do S.O.

Já uma **Task** é um objeto que representa uma única operação, ou seja, uma unidade de trabalho que em algum momento deve ser realizada por uma thread, portanto, geralmente é usada para executar algum código de forma assíncrona. Não é tão custosa como as Threads e foi criada para facilitar a vida do programador, há uma série de ferramentas na API de Task para usar os recursos de forma mais fácil, correta e eficaz, além disso o controle e a comunicação entre as tarefas é muito melhor. Tudo o que precisaria ser feito com threads para o bom uso já está pronto e foi realizado por uma equipe de engenheiros da Microsoft que entende do assunto e teve condições de testar adequadamente.



# 03

## .NET Framework

Um conjunto de recursos e  
funcionalidades para aumentar a  
produtividade do desenvolvedor

# .NET Framework

## O que é .NET Framework?

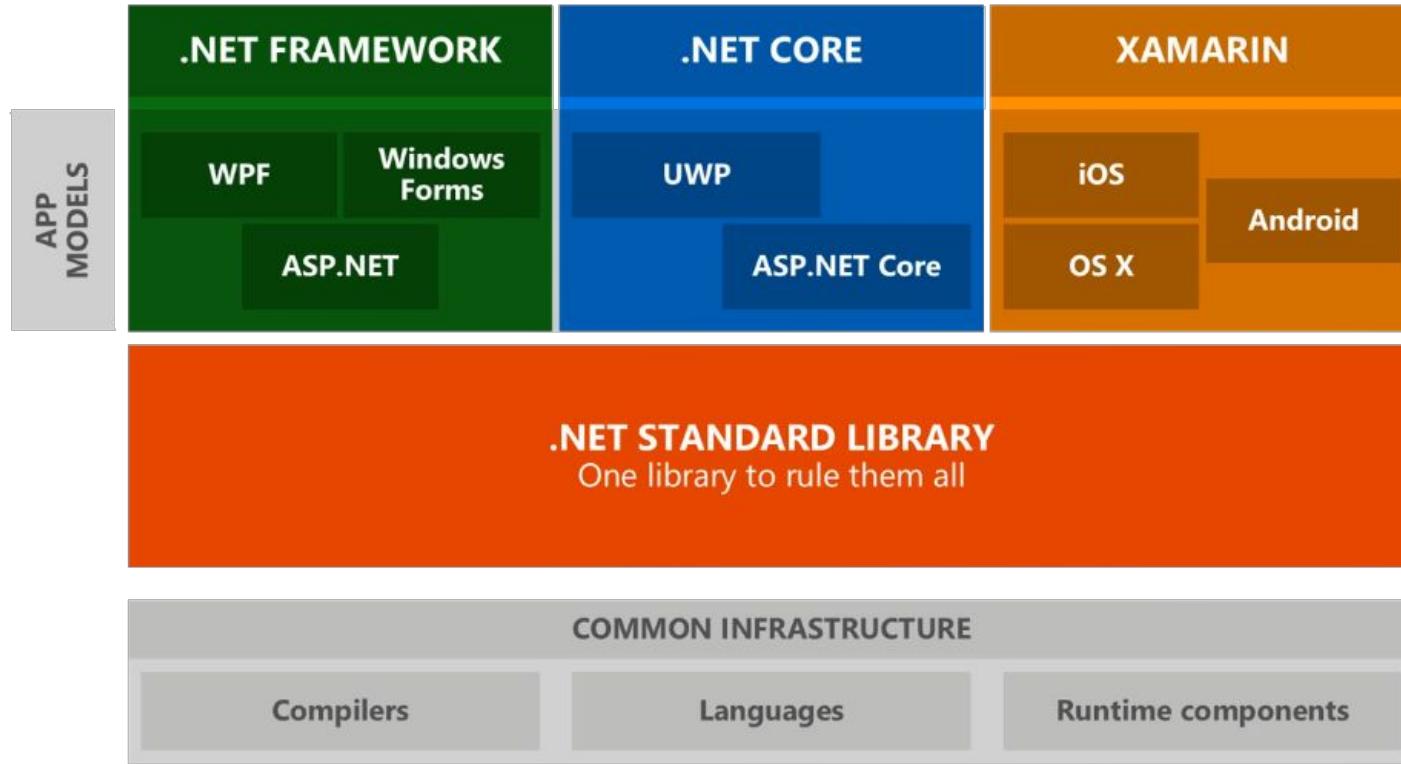
Um framework é uma abstração de funcionalidades que são criadas e distribuídas para que os desenvolvedores possam instalá-lo e usufruir das funcionalidades e do ambiente de compilação que ele propicia.

O .NET Framework é uma iniciativa da empresa Microsoft, que visa criar uma única plataforma de desenvolvimento e execução de sistemas e aplicações. Dessa forma, todo e qualquer código gerado em .NET pode ser executado em qualquer dispositivo que possua o framework instalado.

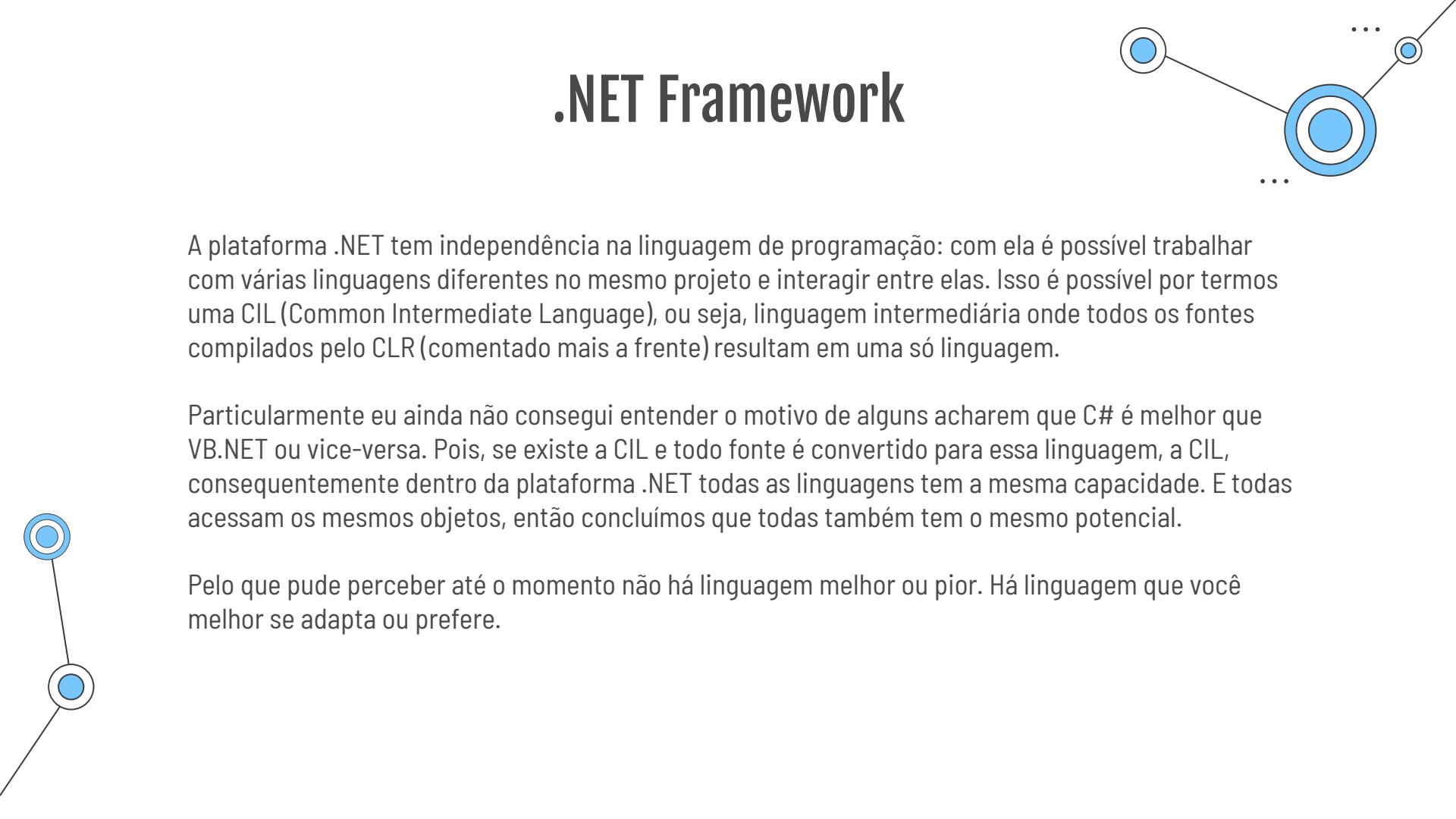
A infraestrutura necessária para executar os códigos escritos para a plataforma .Net é chamada de **CLI (Common Language Infrastructure)**. A CLI engloba a máquina virtual do C# **CLR (Common Language Runtime)** a linguagem intermediária **CIL (Common Intermediate Language)** e os tipos base utilizados nos programas.



# Plataforma .NET



# .NET Framework

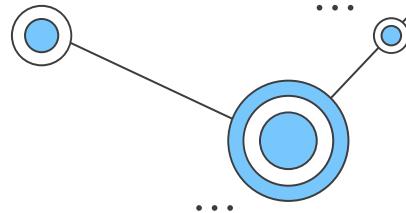


A plataforma .NET tem independência na linguagem de programação: com ela é possível trabalhar com várias linguagens diferentes no mesmo projeto e interagir entre elas. Isso é possível por termos uma CIL (Common Intermediate Language), ou seja, linguagem intermediária onde todos os fontes compilados pelo CLR (comentado mais a frente) resultam em uma só linguagem.

Particularmente eu ainda não consegui entender o motivo de alguns acharem que C# é melhor que VB.NET ou vice-versa. Pois, se existe a CIL e todo fonte é convertido para essa linguagem, a CIL, consequentemente dentro da plataforma .NET todas as linguagens tem a mesma capacidade. E todas acessam os mesmos objetos, então concluímos que todas também tem o mesmo potencial.

Pelo que pude perceber até o momento não há linguagem melhor ou pior. Há linguagem que você melhor se adapta ou prefere.

# Estrutura do .NET



A plataforma .NET possui vários componentes que juntos formam um ambiente muito poderoso, além do CLR e CIL que entraremos em detalhes em outras sessões, vamos conhecer alguns outros componentes e bibliotecas que são muito importante para esse ambiente.

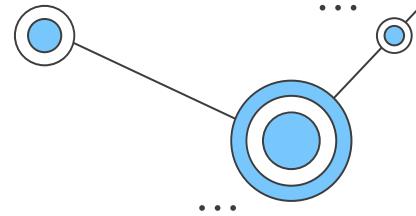
## □ CTS (Common Type System)

É a definição de tipos de dados onde tudo é um objeto e deriva da classe `System.Object`, que é o núcleo do sistema de tipos. Pensando que tudo é um objeto, logo tudo deriva da classe `System.Object` e, por isso, os projetistas da .NET organizaram o sistema de tipos de dados de duas formas:

**Tipos Valor:** variáveis deste tipo são alocadas na pilha e têm como classe base `System.ValueType`, que por sua vez deriva da `System.Object`.

**Tipos Referência:** variáveis deste tipo são alocadas na memória heap e têm a classe `System.Object` como classe base.

# CTS (Common Type System)



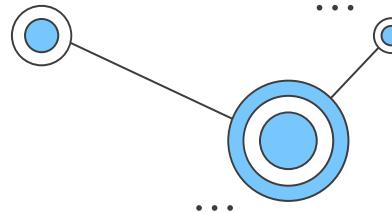
A diferença entre os tipos é que um tipo separa os tipos de estruturas por valor e referencia e o segundo tipo não teria a diferença entre Tipo valor e Tipo referência.

## **System.Object**

- Tipos valor
    - Estruturas
    - Tipos Enumerados
  - Tipo Referência
    - Objeto
    - Interface
    - Ponteiros
- Estruturas
  - Tipos Enumerados
  - Objeto
  - Interface
  - Ponteiros



# Bibliotecas



## **BCL (Base Classe Library - Biblioteca de Classe Base)**

Como o próprio nome diz, na biblioteca de classe base você encontra sistema de janelas, biblioteca de entrada/saída de dados, sockets, gerenciamento de memória, etc.

Esta biblioteca é organizada em uma estrutura conhecida como namespace, ou seja, imagine que você precise atribuir um nome ao seu componente para que o mesmo possa ser referenciado a partir de outro programa. Abaixo seguem alguns namespaces da .NET

### **System:**

Contém algumas classes de baixo nível usadas para trabalhar com tipos primitivos, operações matemáticas, gerenciamento de memória etc.

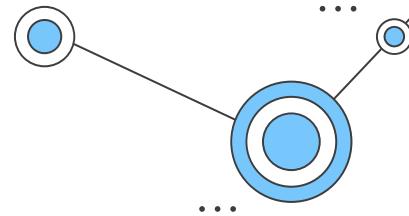
### **System.Collections:**

Contém pilhas, filhas e listas encadeadas.

### **System.Data, System.Data.Common, System.Data.OleDb, System.Data.SqlClient:**

Acesso a base de dados. Aqui também se encontra o ADO.NET

# Bibliotecas



## **System.Diagnostics:**

Log de Event, medição de performance, gerenciamento de processos, depuração etc.

System.Drawing e namespace derivados:

A .NET oferece uma biblioteca de componentes para trabalhar com gráficos, chamadas GDI+, que se encontra neste namespace.

## **System.IO:**

Biblioteca para lidar com entrada e saída, gerenciamento de arquivos etc.

## **System.NET:**

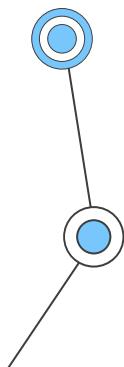
Bibliotecas para programação de redes, sockets etc.

## **System.Reflection:**

Biblioteca necessária para gerar código em tempo de execução, descobrir tipo de variáveis etc.

## **System.Runtime.InteropServices e System.Runtime.Remoting:**

Fornece bibliotecas para interagir com código não-gerenciado.



# Bibliotecas

## **System.XML:**

Biblioteca que permite a interação com documentos XML.

## Estrutura da arquitetura .NET em camadas:

- **Primeira:** Linguagem de programação preferida e aceita pela especificação da CLS e CTS.
- **Segunda:** BCL (Base Class Library)
- **Terceira:** CLR (Common Language Runtime)
- **Quarta:** CTS (Common Type System) e CLS (Common Language Specification).

## **System.Security:**

Criptografia, permissão e todo o suporte referente à segurança.

## **System.Threading:**

Biblioteca para aplicações multithread.

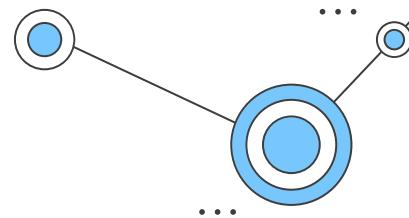
## **System.Web:**

Biblioteca sobre tudo relacionado a Web, como Webservices, ASP.NET etc.

## **System.Windows.Forms:**

Bibliotecas para o desenvolvimento de aplicações Windows tradicionais.

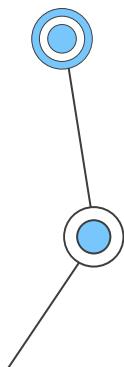
# Bibliotecas



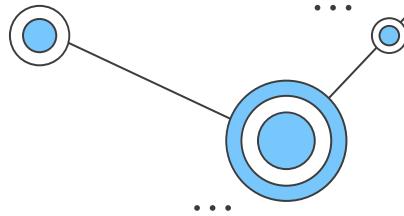
## □ Funcionamento do System.XML:

Explicando melhor as camadas e para que servem, ficaria da seguinte forma:

Você usará sua linguagem de programação preferida e que seja suportada pela .NET (**primeira camada**) e então criar seus sistemas. E, para criá-los, você acessará as classes da BCL, já que tudo é objeto em .NET. (**segunda camada**) Feito isso, seu programa deverá ser compilado e então gerado a IL que, por sua vez, será interpretada pela CLR (**terceira camada**), que deverá passar pelas especificações da CTS e CLS (**quarta camada**).



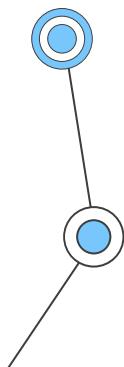
# VES (Virtual Execution System)



É um processo de compilação e é aqui onde o JIT é ativado quando uma aplicação .NET é chamada. O windows identifica que esta é uma aplicação .NET e uma runtime Win32 passa o controle para a runtime do .NET. Neste momento a compilação do PE é efetuada e só então o código assembly próprio da arquitetura do processador é gerado para que a aplicação possa ser executada.

## □ Vamos ver todos os processos em camadas:

- **Primeira:** Sua linguagem de programação
- **Segunda:** CIL
- **Terceira:** CLR (Compilador JIT - Código nativo gerenciado - Execução do código)
- **Quarto:** Dispositivo de saída: Core i5, Snapdragon etc.



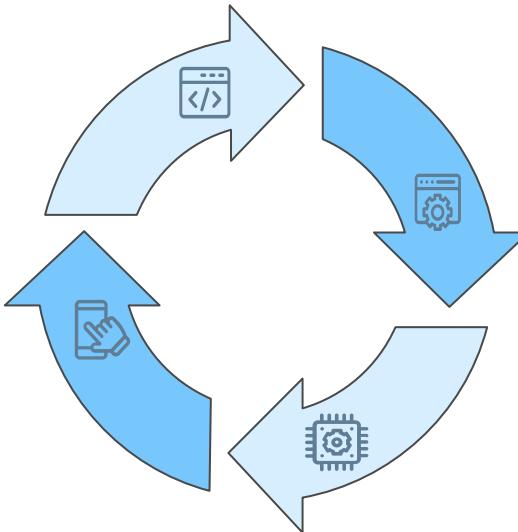
# Processos de Execução em Camadas

## Primeira Camada

Sua Linguagem de Programação

## Quarta Camada

Dispositivo de Saída: Core i5, Snapdragon, etc..



## Segunda Camada

CIL (Common Language Intermediate)

## Terceira Camada

CLR (Compilador JIT)

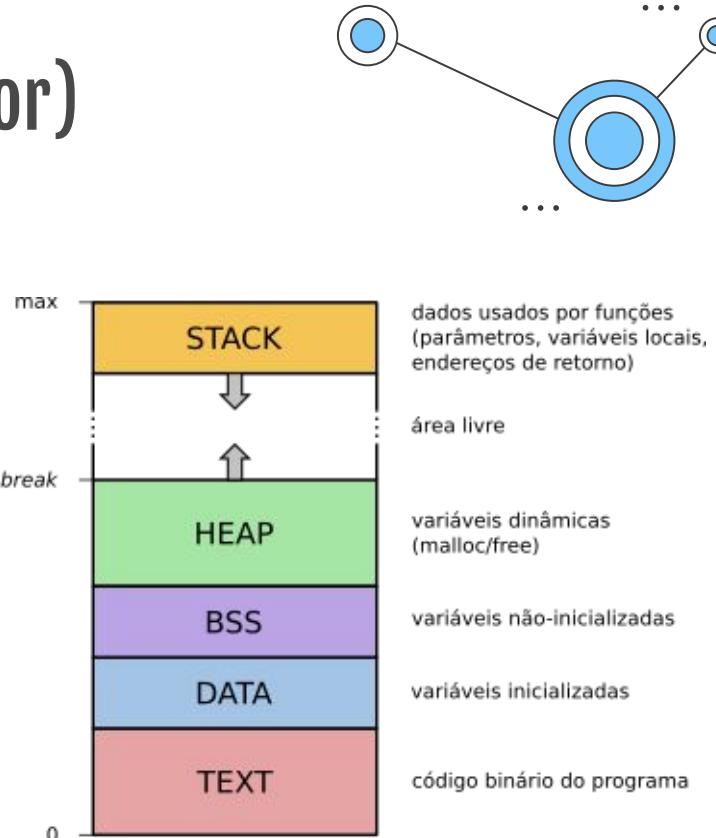
**Em outras palavras as camadas e para que servem ficaria da seguinte forma:**

Você desenvolve seu aplicativo (primeira camada) e compila gerando a CIL (segunda camada). Após isto a CLR interpreta (quando o aplicativo for executado) e compila com o JIT, gerando o código nativo da arquitetura do processador e o executa (terceira camada). De acordo com o dispositivo de saída é gerado um código nativo da arquitetura do processador diferente (quarta camada).

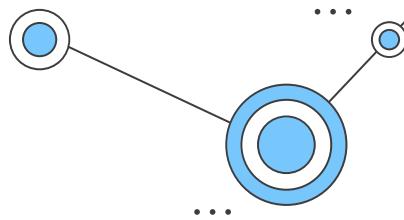
# GC (Garbage Collector)

Devido a complexidade e aos problemas gerados pelo controle manual de memória, o CLR (Common Language Runtime) provê como recurso o Garbage Collector (GC). O GC é um gerenciador automático de memória, responsável por alocar e liberar recursos automaticamente, reduzindo a possibilidade de vazamentos de memória (memory leaks).

Cada processo tem seu próprio espaço de endereçamento virtual. Durante a construção de aplicações usuárias – aquelas que executam no modo usuário, como websites, consoles, aplicativos e serviços, trabalha-se apenas com a manipulação de memória virtual, nunca manipulando diretamente a memória física de um computador.

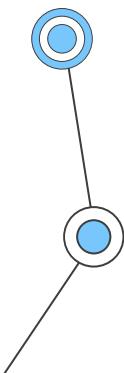


# GC (Garbage Collector)

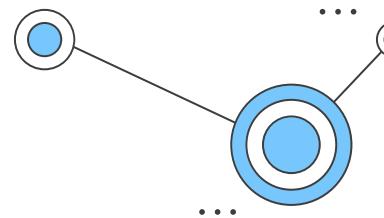


O sistema operacional fica responsável por encapsular a memória física e prover os recursos para que a memória virtual trabalhe corretamente. Cada processo gerenciado contém uma instância própria do GC em execução. Instância cujo trabalho é dedicado à manipulação do espaço de endereçamento virtual de memória do processo ao qual pertence.

O funcionamento do GC é bastante complexo. Nesta seção, serão abordados temas relevantes para o entendimento do modo como o GC funciona e, assim, auxiliar na construção de sistemas mais eficientes.



# GC (Garbage Collector)



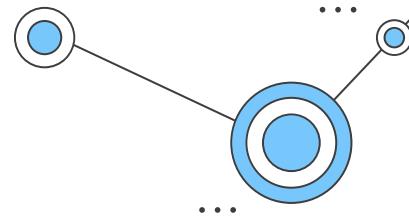
## □ Heap Gerenciado

Depois que uma aplicação gerenciada é iniciada, o CLR inicia uma instância do GC. Ao ser inicializado, o GC aloca dois segmentos de memória virtual. O primeiro deles será utilizado para armazenar e gerenciar objetos ao longo da execução da aplicação. O segundo segmento, chamado de large object heap, é dedicado a “objetos grandes”, considerando objetos grandes aqueles que possuem mais de 85.000 bytes de tamanho.

O heap gerenciado mantém um ponteiro que indica onde o próximo objeto será alocado no heap. A figura a seguir demonstra o heap gerenciado com os objetos A, B e C alocados, além do ponteiro (aqui intitulado de `nextObject`) posicionado onde o próximo objeto deve ser alocado.



# GC (Garbage Collector)



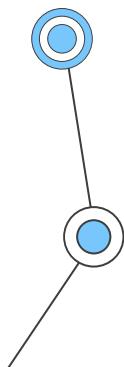
## □ Passos para coleta de memória

O GC avalia quais objetos no heap são usados pela aplicação. Aqueles que não são mais utilizados estão prontos para serem limpados. Porém, o processo de identificação de objetos em uso não é trivial e, por conta disso, é dividido em diversos passos.

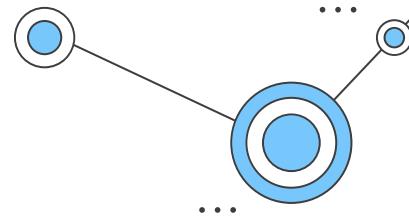
O primeiro deles corresponde ao congelamento de threads. Antes que o GC possa entrar em ação, as threads do processo são congeladas. Assim, nada é executado além do GC durante a coleta de memória. Isso evita que os objetos tenham seus estados alterados enquanto o GC os examina.

Após o congelamento das threads, é iniciado o que chamamos de fase de marcação. No início dessa fase, o GC considera que todos os objetos em memória devem ser limpados. Durante a fase de marcação, o GC varre os roots da aplicação marcando os objetos que estão em uso para, então, excluir aqueles que não estão em uso.

Roots (em português, raízes) são referências a objetos em memória. Eles são estruturas de dados que contém um ponteiro de memória para um objeto de um tipo por referência. Roots podem ser: objetos globais e estáticos, variáveis locais, parâmetros por referência e registradores da CPU.



# GC (Garbage Collector)

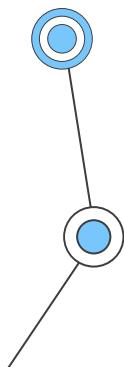


Toda aplicação gerenciada contém uma lista de roots, e é por meio dessa lista que o GC, durante a coleta, constrói um grafo contendo todos os objetos que são acessíveis pelos roots. Objetos acessíveis aos roots incluem os próprios roots, assim como outras instâncias de objetos a que eles fazem referência.

Objetos que estão fora deste grafo são tidos como inacessíveis, sendo assim considerados como fora de uso. Depois de terminada a fase de marcação, o heap contém um conjunto de objetos marcados e um conjunto de objetos desmarcados. Os objetos desmarcados são descartados, e então é iniciada a fase de compactação.

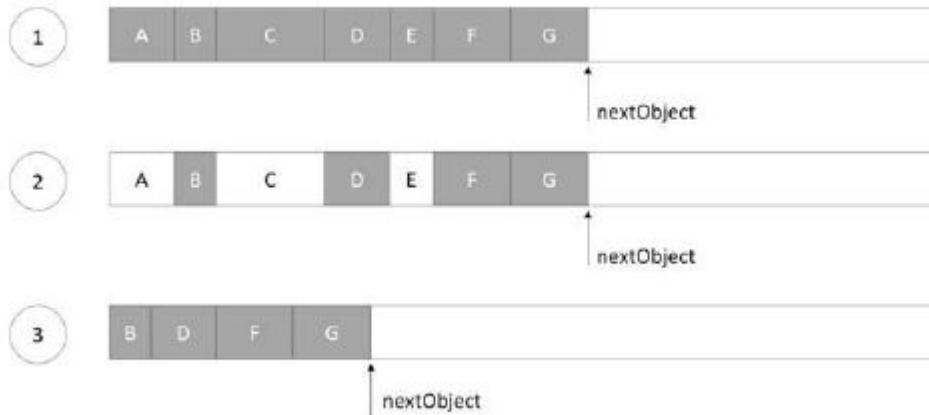
Na fase de compactação, o GC percorre o heap gerenciado procurando por blocos contínuos de objetos desmarcados. Se blocos pequenos são encontrados, eles são ignorados. Se blocos grandes são encontrados, então os objetos são deslocados para que o heap seja compactado, reduzindo a fragmentação.

Ao deslocar os objetos na memória heap, seus endereços de memória são alterados, invalidando qualquer referência a eles. Essa compactação obriga o GC a percorrer todos os objetos atualizando seus ponteiros de memória para os novos endereços dos objetos deslocados.

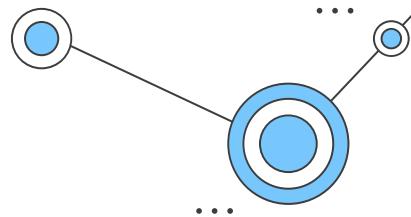


# GC (Garbage Collector)

Essa figura demonstra as alterações no heap gerenciado durante as fases de coleta do GC. Na primeira ilustração do heap, temos os objetos alocados em memória prontos para o início da coleta. Na segunda ilustração, temos os objetos em uso marcados e os objetos que não estão em uso desmarcados. E, por fim, na última ilustração, temos o heap gerenciado compactado e com apenas objetos em uso pela aplicação ativos em memória.



# GC (Garbage Collector)



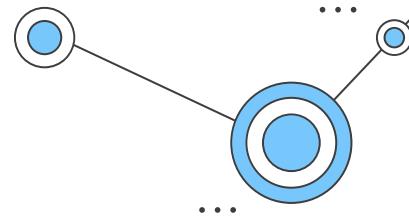
## □ Cenários de coleta e gerações

Quanto menor a quantidade de objetos no heap gerenciado, menor será o trabalho do GC. Existem situações que estimulam a coleta de memória do GC, sendo elas:

- O sistema tem pouca memória física e essa pressão de memória exige a liberação de recursos;
- A quantidade de memória alocada no heap gerenciado ultrapassa seu limite, e uma liberação de recursos é necessária para alocação de mais objetos;
- O método `System.GC.Collect` é executado e, assim, o GC é levado a coleta prematura de recursos;
- O CLR é desligado. Isso acontece quando um processo termina normalmente.



# GC (Garbage Collector)



## □ Detalhes do Algoritmo de Coleta do GC:

Como prática para otimização do seu trabalho, o GC do .NET Framework organiza a pilha de objetos em gerações de objetos de longa e de curta duração. Esse modelo de algoritmo de garbage collector, baseado em gerações (também conhecido como garbage collector efêmero), possui alguns pressupostos:

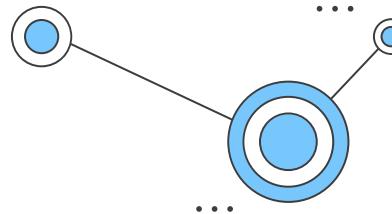
- Quanto mais jovem um objeto for, mais curto será seu tempo de vida.
- Quanto mais velho um objeto for, mais longo será seu tempo de vida.
- Coletar porções do heap é mais rápido do que coletar o heap inteiro.



## □ No .NET Framework, o GC utiliza três gerações, sendo elas:

- **Geração 0:** esta é geração que mais sofre coletas de recursos, pois corresponde a geração de objetos de curta duração. Variáveis temporárias e variáveis com escopo de método são as principais candidatas a estarem na Geração 0.
- **Geração 1:** buffer intermediário de objetos entre a Geração 0 e a Geração 2.
- **Geração 2:** contém objetos de longa duração. São tidos como objetos de longa duração aqueles objetos que contêm conteúdo estático (static), ou que são instâncias ativas durante toda a execução do processo.

# GC (Garbage Collector)



Quando iniciado, o heap gerenciado não contém objetos, e todos novos objetos são adicionados na Geração 0. Objetos que estão na Geração 0 são objetos que foram recentemente criados e que nunca sofreram uma coleta do GC, como demonstrado na figura a seguir.

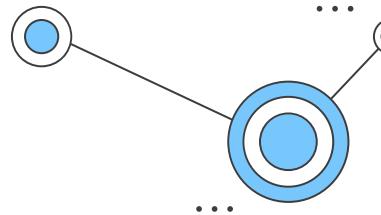
No exemplo desta figura, demonstra-se um conjunto de objetos sendo criados no heap gerenciado sem qualquer execução prévia da coleta de memória do GC.



Se o limite de alocação da Geração 0 for atingido, o GC é acionado para executar sua primeira coleta de memória. Após remover todos os objetos que não estavam mais em uso, os objetos restantes (que são tidos como em uso pela aplicação) são promovidos para Geração 1, ficando vazia a coleção de objetos da Geração 0, conforme a figura seguinte.



# GC (Garbage Collector)



Por padrão, durante a continuidade da execução da aplicação, novos objetos serão alocados. E estes novos objetos serão alocados na Geração 0, como apresentado na figura adiante. Nota-se que a Geração 1 mantém-se intocada, contém apenas os objetos sobreviventes da primeira coleta do GC.

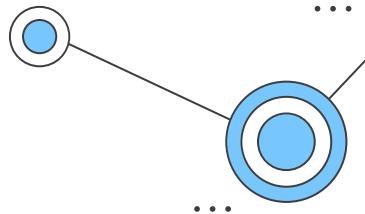


Quando sob pressão (ou seja, quando a cota de alocação da Geração 0 for atingida), novamente o GC iniciará sua coleta de memória. Nesse instante, o GC deve decidir quais gerações serão examinadas. Se a Geração 1 ainda estiver dentro do seu limite de consumo de memória, ela não será examinada.

Assim, apenas a Geração 0 será alvo da coleta, liberando instâncias de objetos que não são usadas e promovendo as instâncias que ainda estão em uso para a Geração 1, como demonstrado na figura:



# GC (Garbage Collector)



Como esperado, a aplicação continuará a criar novas instâncias de objetos, preenchendo a Geração 0 com essas novas instâncias.

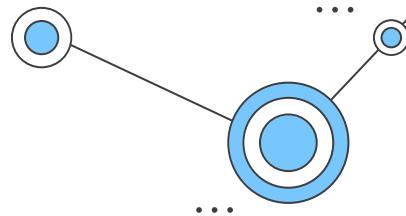


Sob nova pressão de memória, o GC vai novamente decidir quais gerações serão examinadas. Caso a Geração 1 esteja acima do seu limite de memória, ela também será examinada, assim como a Geração 0. Se, durante a execução da aplicação, objetos contidos na Geração 1 deixarem de ser utilizados, então serão eliminados da memória, seguindo as mesmas regras da Geração 0.

Os objetos da Geração 1 que sobreviverem à coleta de memória serão promovidos para Geração 2, enquanto a Geração 0, ao fim de cada coleta, está livre de objetos gerenciados.



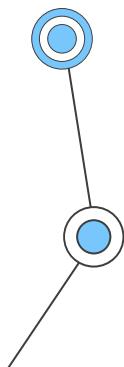
# GC (Garbage Collector)



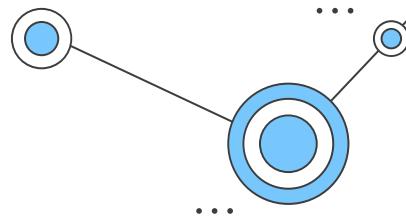
Objetos contidos na Geração 2 sobreviveram a duas ou mais coletas do GC. Como mencionado anteriormente, o algoritmo de garbage collector do .NET Framework contém apenas três gerações.

O Garbage Collector é um gerenciador de memória que se auto ajusta para melhor adaptar sua performance. As três gerações iniciam com limites de alocação de memória que se adaptarão ao longo de sua execução para melhor adequar o funcionamento do GC.

Por exemplo, suponha que a Geração 0 inicia com limite de memória de 256Kb. Se, durante sua execução, o GC notar que poucos objetos estão sobrevivendo a sua coleta na Geração 0, ele pode reduzir o limite de memória dessa geração para 128Kb. Da mesma maneira, se o GC avaliar que muitos objetos sobrevivem à coleta da Geração 0, ele pode adaptar seu limite para 512Kb. Automaticamente, aumenta-se o limite de memória da geração e diminui-se a quantidade de varreduras, pois quanto maior o limite de memória de uma geração, menor será a quantidade de vezes que o GC vai examiná-la.



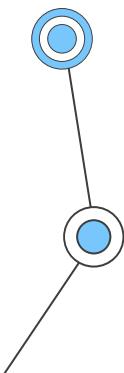
# GC (Garbage Collector)



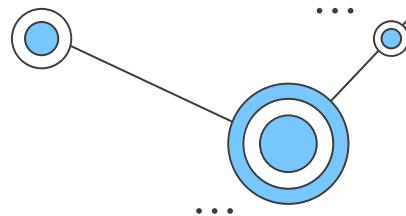
O GC pode utilizar essa mesma heurística para manipular os limites das Gerações 1 e 2. Os limites de tamanho de cada geração são adaptáveis a plataforma que está em execução.

O uso da estratégia de gerações no algoritmo do GC melhora inclusive a performance da fase de marcação, pois se um objeto pertencer a uma geração mais antiga, todos os objetos para os quais aquele faz referência são ignorados. Dessa forma, não há necessidade de varrer todos os objetos contidos no heap gerenciado.

Entretanto, é possível que um objeto antigo refcrcie um objeto mais novo. Neste cenário, o GC usa um mecanismo interno do compilador JIT (Just-In-Time), que marca os objetos que tiveram mudanças em seus campos. Somente esses objetos marcados são examinados.



# GC (Garbage Collector)



## □ Large Object Heap

O CLR considera um objeto como sendo, um objeto pequeno, ou um objeto grande. Entende-se por objetos grandes aqueles que possuem mais de 85.000 bytes ou 83Kb de tamanho.

Esses objetos são tratados de forma um pouco diferente daqueles considerados pequenos:

- Objetos grandes não são armazenados dentro do mesmo espaço de endereçamento que os pequenos. São armazenados em outra área do espaço de endereçamento do processo.
- Esses objetos não sofrem compactação, pois haveria um alto custo de performance para movê-los na memória. Por isso, a fragmentação pode acontecer na estrutura desses objetos.
- São imediatamente considerados como estando na Geração 2. Devido a isso, esses tipos de objetos devem ser criados se a intenção é mantê-los em uso por um longo período.

Objetos grandes são, por exemplo, strings grandes (como um arquivo XML) ou arrays usados em operações de I/O.



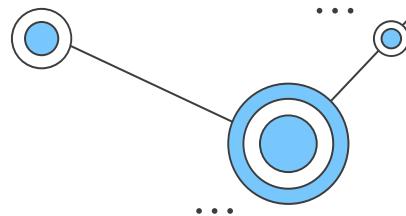
# GC (Garbage Collector)

## □ Modelos de execução do Garbage Collector

Existem dois modelos de execução do garbage collector do .NET Framework, são eles: **workstation mode** e **server mode**.

- O **workstation mode** é o modelo padrão de execução do GC. Este pressupõe que a aplicação executa em uma máquina na qual outras aplicações usuárias concorrem por CPU e, por conta disso, não expõe todo seu poder de processamento. Neste modelo, o GC cria apenas um heap gerenciado para todo o consumo de memória da aplicação, além de um thread para gerenciar a alocação e limpeza de memória.
- O **server mode** é o modelo que aperfeiçoa o GC para execução de aplicações em servidores. Neste modelo, é criado um heap gerenciado, além de um thread para administração da memória, para cada núcleo de processamento lógico. Assim, em um ambiente multiprocessado com, por exemplo, quatro núcleos de processamento, a aplicação terá quatro heaps gerenciados e quatro threads para gerenciar a alocação de memória. Com múltiplos threads trabalhando juntos em paralelo, a coleta de memória é muito mais rápida se comparada à execução com workstation mode.

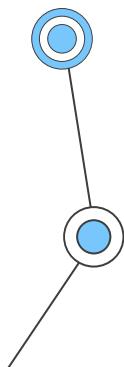
# GC (Garbage Collector)



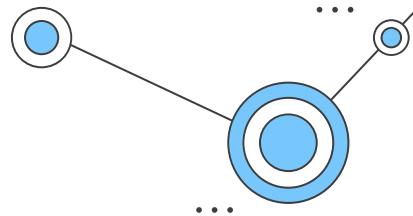
## □ Método Finalize

Alguns objetos requerem mais do que simplesmente memória para serem úteis encapsulando, por exemplo, recursos nativos como arquivos, conexões com o banco de dados, sockets etc. O CLR sabe lidar bem com objetos alocados no heap gerenciado, porém nada sabe sobre os recursos nativos. Consequentemente, ao ocorrer um GC, o objeto gerenciado é expurgado corretamente, enquanto o recurso nativo não é liberado, podendo causar vazamentos de memória.

Devido a essa particularidade, é necessário que a limpeza ou liberação do recurso aconteça antes que o objeto que o encapsula tenha sua memória recuperada. Isso é possível por meio de uma funcionalidade provida pelo CLR, conhecida como finalização.



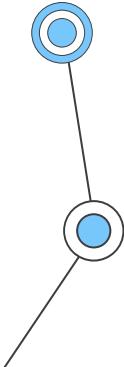
# GC (Garbage Collector)



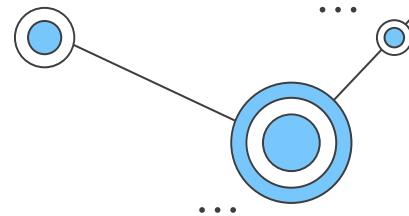
A finalização ocorre em um objeto quando este chama o método **Finalize**. Este pode ser sobreescrito e é definido usando uma sintaxe especial, colocando o símbolo til (~) na frente do nome da classe, como mostra o exemplo:

## *Definindo o método Finalize*

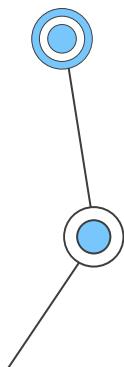
```
public class MinhaClasse {  
    // Esse é o método Finalize.  
    ~MinhaClasse() {  
        // Código para a limpeza de recursos.  
    }  
}
```



# GC (Garbage Collector)

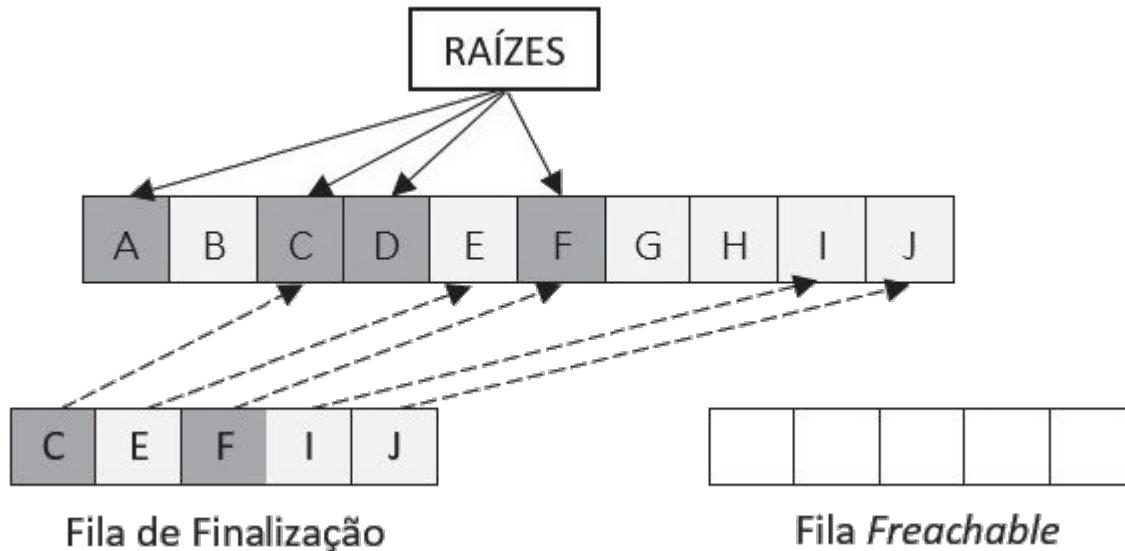


- **Ao projetar uma classe, porém, é melhor evitar usar o método Finalize pelas seguintes razões:**
  - Esse tipo de objeto leva mais tempo para ser alocado.
  - Objetos finalizáveis são promovidos para gerações mais antigas, prevenindo de serem coletados assim que o GC determina que estes devem ser expurgados. Além de1 que todos os objetos relacionados a eles também são promovidos de geração, forçando estes a viverem muito mais do que o necessário.
  - Não há controle sobre quando o método Finalize será executado, e os recursos podem ser mantidos até a próxima execução do GC, aumentando, assim, a pressão na memória.
  - Não há garantia quanto a ordem de execução com que os métodos Finalize serão chamados. Por isso, é preciso evitar que se façam referências, de dentro do método, a objetos que também implementam o **Finalize** pois esses podem já ter sido finalizados.



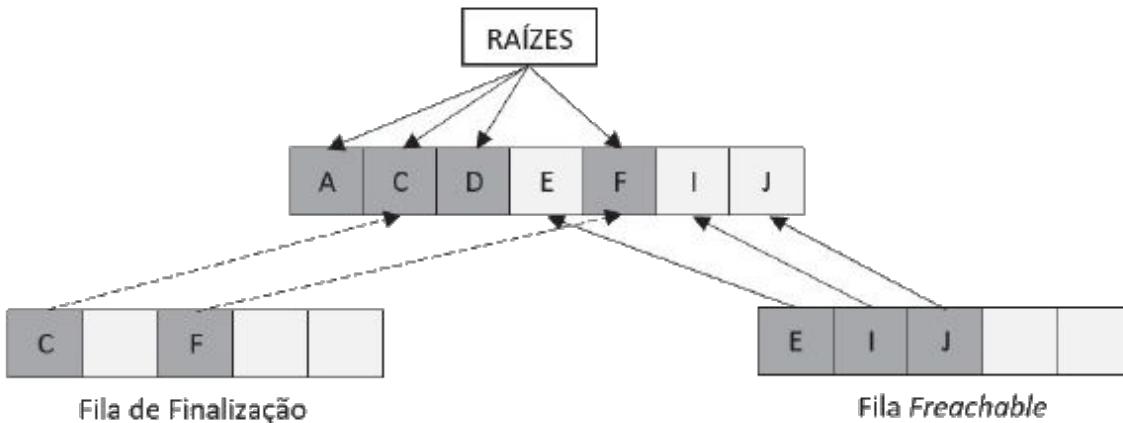
# GC (Garbage Collector)

Ao instanciar um objeto da classe `MinhaClasse`, por ter definido o método `Finalize`, um ponteiro para esse objeto é adicionado na fila de finalização. Essa fila é uma estrutura interna usada pelo GC que mantém os endereços dos objetos que precisam de uma limpeza adicional antes de serem coletados.

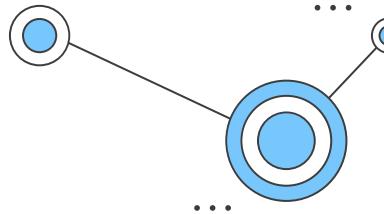


# GC (Garbage Collector)

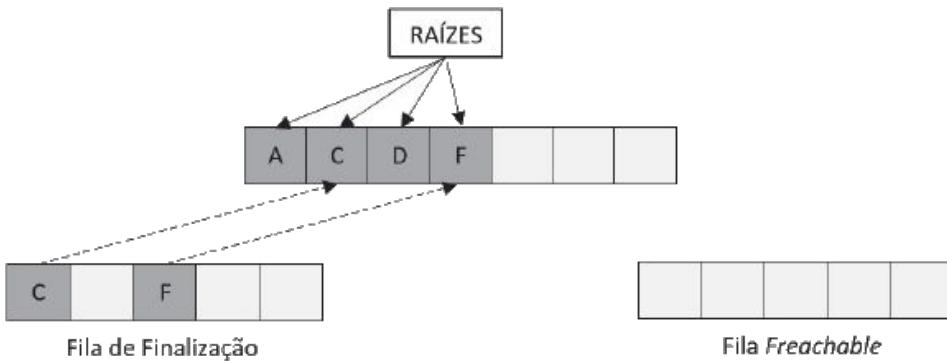
Ao ocorrer o GC, os objetos B, E, G, H, I e J são marcados para terem suas memórias coletadas. A fila de finalização é, então, analisada em busca de referências a esses objetos. Ao encontrá-las, elas são removidas da fila e são adicionadas na fila freachable. Neste momento, os objetos referenciados por esta fila são trazidos de volta a "vida" (daí o nome reachable - alcançável - da fila), pois eles precisam ser acessíveis para que sejam finalizados. Esse fenômeno é conhecido como ressureição. Importante ressaltar que, após a fase de compactação do GC, as referências ainda contidas na fila de finalização têm seus valores atualizados caso os objetos para os quais apontam tenham sido movidos na memória.



# GC (Garbage Collector)



Como é possível notar nessa figura, os objetos E, L e J sobreviveram ao GC, logo, sendo promovidos para gerações mais antigas. Após terem seus métodos Finalize executados, os objetos têm suas referências removidas da fila freachable e estão prontos para serem coletados. Assim, conclui-se que são necessárias de, no mínimo, duas execuções do GC para que suas memórias sejam recuperadas.



# GC (Garbage Collector)

## Conclusão

Então resumindo esse conteúdo extenso sobre **GC (Garbage Collector)** ele é um processo usado para a automação do **gerenciamento de memória**. Com ele é possível recuperar uma área de memória inutilizada por um **software**, o que pode evitar problemas de vazamento de memória, resultando no esgotamento da memória livre para alocação.

Esse sistema é o oposto do gerenciamento manual de memória, em que o desenvolvedor deve especificar explicitamente quando e quais objetos devem ser **desalocados** e retornados ao sistema, entretanto, muitos sistemas usam uma combinação das duas abordagens.

Os princípios básicos do GC são encontrar objetos de um programa que não serão mais acessados no futuro, e deslocar os recursos utilizados por tais objetos. Tornando a desalocação manual de memória desnecessária, e geralmente proibindo tal prática, o coletor de lixo livra o programador de se preocupar com a liberação de recursos já não utilizados, o que pode consumir uma parte significativa do ciclo de desenvolvimento do software. Também evita que o desenvolvedor introduza erros no programa devido a má utilização de **ponteiros**.

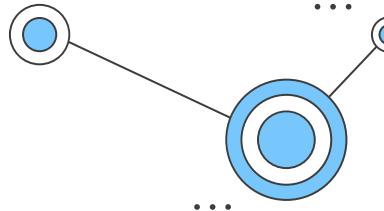
# HistÓria do C# e .NET Framework

Entender um pouco da história do C# e do .Net é essencial para enxergar os motivos que a levaram ao sucesso.

No final da década de 1990 a Microsoft tinha diversas tecnologias e linguagens de programação para resolver muitos problemas diferentes. Toda vez que um programador precisava migrar para uma nova linguagem, era necessário aprender tanto a nova linguagem quanto suas bibliotecas e conceitos. Para solucionar esses problemas, a Microsoft recorreu à linguagem Java.

O Java agradou os engenheiros da Microsoft pois com ela podíamos construir programas que eram independentes do ambiente de execução, além de possuir diversas bibliotecas com soluções prontas para diversos problemas. Para lançar produtos baseados no Java, a Microsoft assinou um acordo de licenciamento com a Sun para utilizar o Java em ambiente Windows.

# HistÓria do C# e .NET Framework



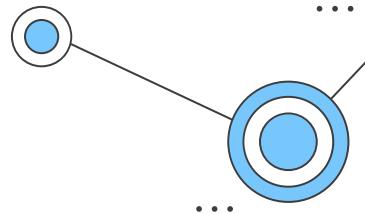
Porém, a linguagem Java possuía um grave problema: ela não se comunicava bem com as bibliotecas de código nativo (código de máquina) que já existiam. Para resolver isso, a Microsoft decidiu criar a sua própria implementação do Java chamado J++, que possuía extensões proprietárias que resolviam o problema de comunicação com o código nativo existente. Para o desenvolvimento dessa nova implementação do Java, a Microsoft contratou um engenheiro chamado **Anders Hejlsberg**, um dos principais nomes por trás do Delphi.



O J++ era uma versão da linguagem Java que só podia ser executada no ambiente Microsoft. Seu código não podia ser executado em mais nenhum ambiente Java, o que violava o licenciamento feito com a Sun e, por isso, a Microsoft foi processada. Uma das mais conhecidas batalhas judiciais da época.

Sem o J++, a Microsoft foi obrigada a repensar sua estratégia sobre como lidar com as diferentes linguagens e tecnologias utilizadas internamente. A empresa começou a trabalhar em um novo plataforma que seria a base de todas as suas soluções, que posteriormente foi chamada de **.NET**. Esse novo ambiente de desenvolvimento da Microsoft foi desde o início projetado para trabalhar com diversas linguagens de programação, assim diversas linguagens diferentes compartilhariam o mesmo conjunto de bibliotecas. Com isso, para um programador migrar de uma linguagem para outra ele precisaria apenas aprender a linguagem sem se preocupar com as bibliotecas e APIs.

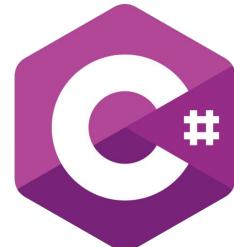
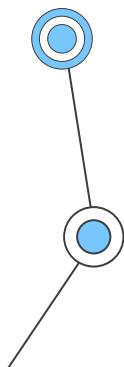
# HistÓria do C# e .NET Framework



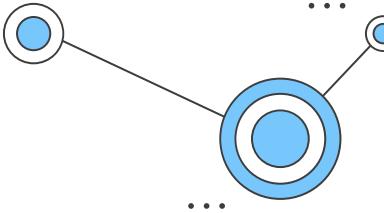
Além de uma plataforma a Microsoft também precisava de uma linguagem de programação. Um novo projeto de linguagem de programação foi iniciado, o projeto COOL (C-like Object Oriented Language).

Anders Hejlsberg foi escolhido como engenheiro chefe desse novo projeto. COOL teve seu design baseado em diversas outras linguagens do mercado como Java, C, C++, Smalltalk, Delphi e VB. A ideia era estudar os problemas existentes e incorporar soluções.

Em 2002, o projeto COOL foi lançado como linguagem C# 1.0, junto com o ambiente .Net 1.0. Atualmente, a linguagem C# está em sua versão 10, e o .Net na versão 6, tendo evoluído com expressiva velocidade, adotando novidades na sua sintaxe que a diferenciam bastante do Java e outras concorrentes.



# Maquina Virtual



Em uma linguagem de programação como C e Pascal, temos a seguinte situação quando vamos compilar um programa:

O código fonte é compilado para código de máquina específico de uma plataforma e sistema operacional. Muitas vezes o próprio código fonte é desenvolvido visando uma única plataforma!



Esse código executável (binário) resultante será executado pelo sistema operacional e, por esse motivo, ele deve saber conversar com o sistema operacional em questão. Isto é, temos um código executável diferente para cada sistema operacional diferente. Precisamos reescrever um mesmo pedaço da aplicação para diferentes sistemas operacionais, já que eles não são compatíveis.

O C# utiliza o conceito de **máquina virtual**. Entre o sistema operacional e a aplicação existe uma camada extra responsável por "traduzir" – mas não apenas isso – o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional onde ela está rodando no momento.

# CLR (Common Language Runtime)

Repare que uma máquina virtual é um conceito bem mais amplo que o de um interpretador. Como o próprio nome diz, uma máquina virtual é como um "computador de mentira": tem tudo que um computador tem. Em outras palavras, ela é responsável por gerenciar memória, threads, a pilha de execução etc.

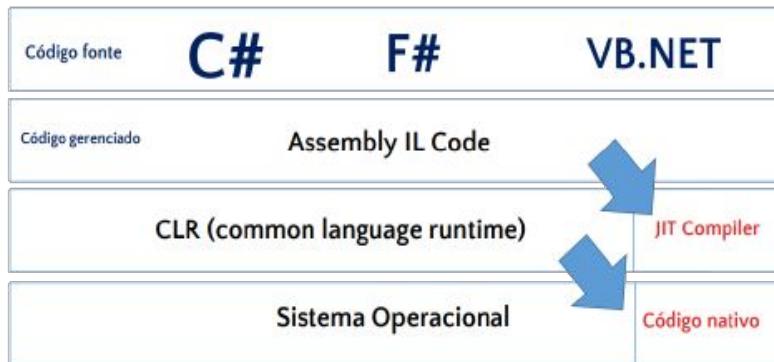
Sua aplicação roda sem nenhum envolvimento com o sistema operacional! Sempre conversando apenas com a máquina virtual do C#, a **Common Language Runtime (CLR)**. O trabalho da CLR é executar diversas linguagens, tais como C#, F# e VB.NET e permitir grande interoperabilidade entre elas. Além disso, fornece serviços adicionais, incluindo gerenciamento de memória, segurança de tipagem, tratamento de exceção, garbage collection, segurança e gerenciamento de thread. Todos os programas escritos para o framework .NET, independentemente da linguagem de programação, são executados pela CLR. Todas as versões do framework .NET incluem a CLR.

O CLR isola totalmente a aplicação do sistema operacional. Se uma aplicação rodando no CLR termina abruptamente, ela não afetará as outras máquinas virtuais e nem o sistema operacional. Essa camada de isolamento também é interessante quando pensamos em um servidor que não pode se sujeitar a rodar código que possa interferir na boa execução de outras aplicações.

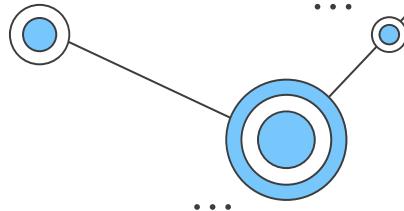
# CIL (Common Intermediate Language)

Como a máquina virtual deve trabalhar com diversas linguagens de programação diferentes, a CLR não pode executar diretamente o código do C#, ela precisa executar uma linguagem intermediária comum a todas as linguagens da plataforma .Net, a **CIL (Common Intermediate Language)**. A CIL é uma linguagem de programação de baixo nível do ambiente de programação da Microsoft. O código de mais alto nível do ambiente .NET Framework é compilado em código CIL, que é assemblerado em código chamado bytecode.

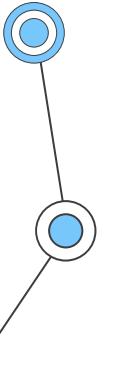
Para gerar o CIL que será executado pela CLR, precisamos passar o código C# por um compilador da linguagem, como o programa csc.exe. O compilador lê o arquivo com o código fonte do programa e o traduz para o código intermediário que será executado pela máquina virtual.



# JIT (just-in-time)



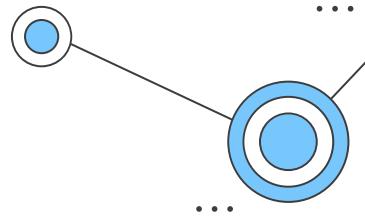
Para executarmos uma aplicação C#, precisamos passar o código CIL do programa para a CLR, a máquina virtual do .NET. A CLR por sua vez precisa executar o código da aplicação no sistema operacional do usuário e, para isso, precisa emitir o código de máquina correto para o ambiente em que o programa está sendo executado. Mas a CLR não interpreta o CIL do programa, isso seria muito lento, ao invés disso, quando o programa C# é carregado na memória, a CLR converte automaticamente o código CIL para código de máquina, esse processo é feito por um compilador **Just in Time (JIT)** da CLR.



O JIT nada mais é que uma compilação feita em tempo de execução, ao invés de antes de rodar a aplicação. Ela pode ser feita por arquivo, função ou até fragmentos de código, traduzindo dinamicamente essas partes e executando diretamente na memória. ou seja, o JIT mescla os dois tipos de tradutores. Ele compila apenas parte do código que será usado na execução e interpreta essa fração. Os ganhos de performance são notáveis. Mas essa técnica ainda apresenta algumas desvantagens como o atraso na inicialização, pois ainda é necessário carregar os primeiros blocos do código para serem compilados. Assim, quanto mais o JIT for otimizado, melhor o código gerado, mas também esse atraso fica maior.

Esse carregamento utilizando o JIT faz com que o código escrito na linguagem C# execute com o desempenho máximo, o mesmo de um programa escrito em linguagens que compilam diretamente para o código de máquina, mas com a vantagem de executar no ambiente integrado do .NET.

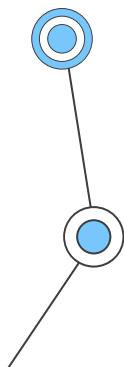
# Conclusão



## Fluxo de Execução

Quando pedimos para o Visual Studio executar uma aplicação, ele chama o compilador da linguagem C# passando os arquivos de texto que contém o código da aplicação (código fonte do programa). Caso o código fonte não tenha nenhum erro de sintaxe, o compilador gera o código intermediário (**CIL, Common Intermediate Language**) que é entendido pela máquina virtual da linguagem C#, a **CLR (Common Language Runtime)**. O código CIL é colocado em um arquivo executável (arquivo com extensão .exe) dentro da pasta do projeto. Esse arquivo que é resultado da compilação do programa é chamado de Assembly dentro da linguagem C#.

Depois da compilação, o Visual Studio executa o assembly gerado na máquina virtual do C#. A CLR por sua vez carrega o código CIL que foi gerado pelo compilador e o executa no sistema operacional, mas se a CLR interpretasse o código CIL para linguagem de máquina, o desempenho do C# não seria muito bom, e por isso, quando um programa C# é carregado pela CLR ele já é automaticamente convertido para linguagem de máquina por um processo conhecido como **JIT (Just-in-time)**. Então no C#, o código sempre é executado com o mesmo desempenho do código de máquina.



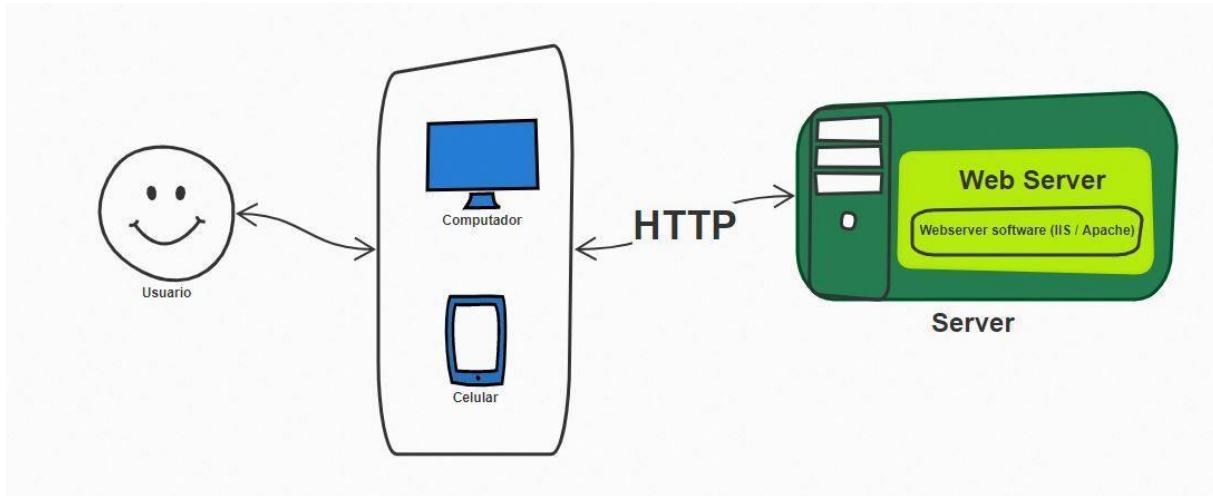
# 04

## Servidor Web

Uma forma de expor sua aplicação  
para o mundo

# Introdução

O **IIS** é um **WebServer**. Um software que expõe um site na internet através do protocolo HTTP. Basicamente, um WebServer é um software que permite que recursos (páginas da web, imagens, etc.) sejam solicitados através do protocolo HTTP. Um WebServer é um "computador normal". Que executa um software especial que o torna um WebServer. Que nesse caso é o IIS. A grosso modo é possível dizer que o IIS é um programa que aceita conexões HTTP.

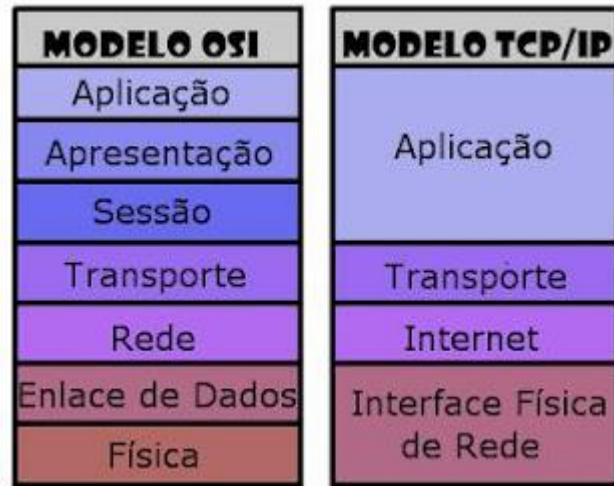


# Protocolo TCP/IP

Antes de falar sobre os componentes da arquitetura do IIS é necessário entender como funciona o **protocolo TCP/IP**.

Um **protocolo** é um conjunto de regras que possibilita a comunicação entre dois ou mais dispositivos.

O modelo TCP/IP, quando comparado com modelo OSI, tem duas camadas que se formam a partir da fusão de algumas camadas, são elas: as camadas de Aplicação ( aplicação, apresentação e sessão ) e Redes ( link de dados e física ).



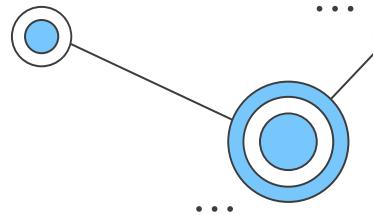
# Protocolo TCP/IP

Conforme vemos na figura, o TCP/IP possui quatro níveis de processamento de dados. Assim como no OSI, os programas do computador se comunicam diretamente com o nível “**aplicação**”.

## □ É nessa camada que estão os protocolos:

- **SMTP (Simple Mail Transfer Protocol)**, ou protocolo de transferência simples de e-mail. Responsável pela decodificação de e-mails.
- **FTP (File Transfer Protocol)**, ou protocolo de transferência de arquivos.
- **HTTP (Hypertext Transfer Protocol)**, ou protocolo de transferência de hipertexto. Protocolo que permite a navegação na Internet.

# Protocolo TCP/IP



Cada protocolo recebe as informações específicas do programa pelo qual ele é responsável. Por exemplo, se você quer entrar na Internet, ao abrir o navegador, esse programa se comunicará diretamente com o protocolo HTTP.

Se quiser enviar um e-mail através do seu programa de envio de e-mail, ele mandará os dados para o protocolo SMTP.

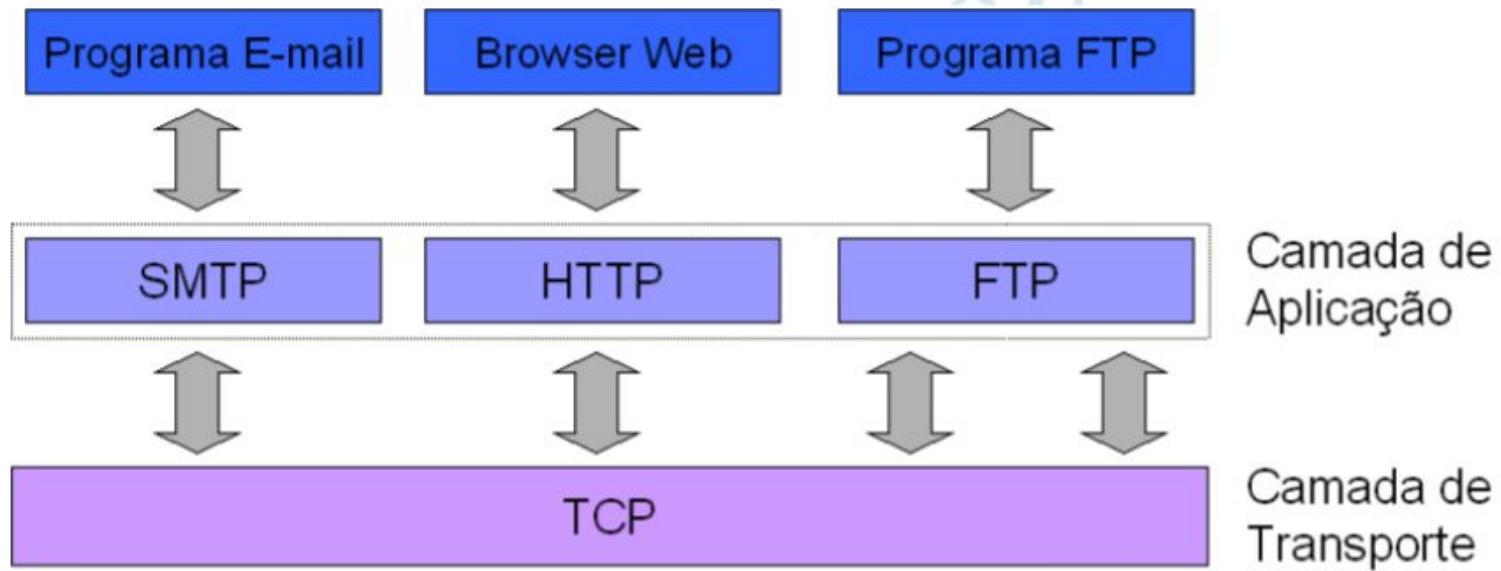
Depois que o protocolo específico para a solicitação do programa processa o dado enviado, o **nível aplicação** entra em contato com a **camada transporte**, que nesse caso é constituída do protocolo TCP.

## As funções do TCP são:

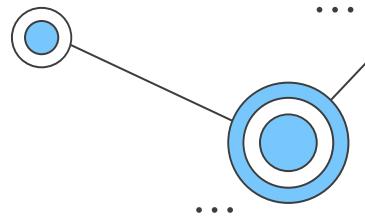
- Dividir os dados em pacotes.
- Enviar os pacotes para a **camada inferior (Internet)**.
- Quando o computador está recebendo informações, o TCP organiza os pacotes.
- Também verifica se os dados contidos nos pacotes chegaram em ordem e em perfeito estado, ou seja, se não foram corrompidos durante a transferência.

# Protocolo TCP/IP

- Resumidamente, a comunicação do programa ou do aplicativo do computador é até chegar ao protocolo TCP é a seguinte:



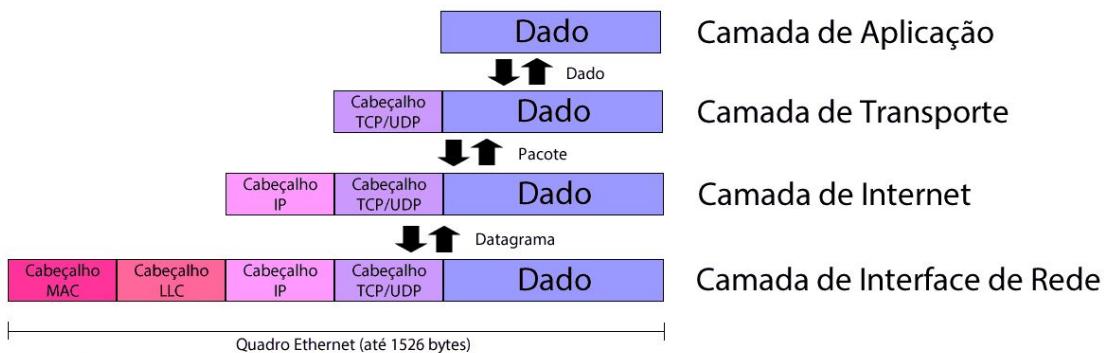
# Protocolo TCP/IP



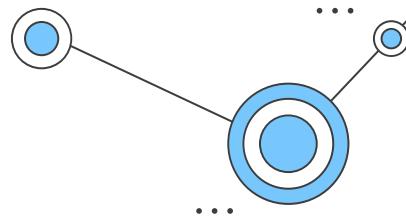
Depois que o TCP fez seu trabalho, os dados são transmitidos para a **camada Internet**, que coloca o endereço de envio das informações nos pacotes divididos pela camada superior, assim como identifica o computador que está enviando as informações.

Os endereços virtuais (de destino e de envio) são conhecidos como **"endereços IP"**.

Quando o IP está identificando o pacote de dados, as informações são transmitidas para a camada interface com a rede, que faz um trabalho análogo ao nível físico do modelo de referência OSI, ou seja, transforma os pacotes em meios físicos de transmissão pela rede.

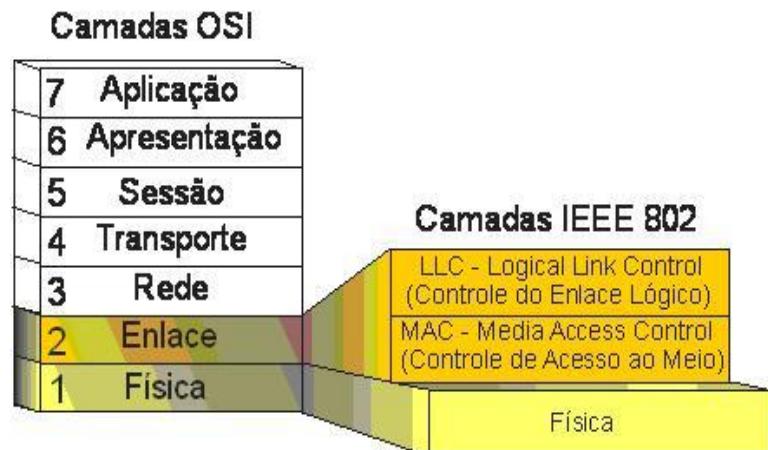


# Protocolo TCP/IP



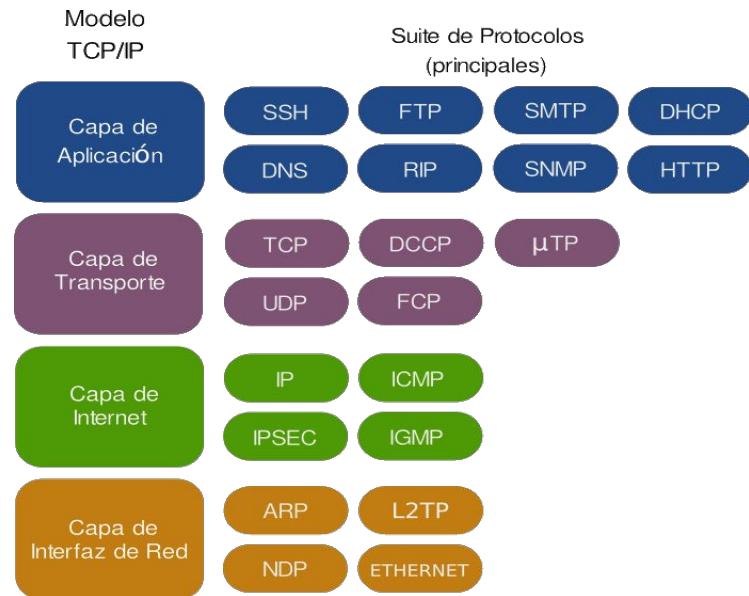
Perceba, na figura, que, a cada camada que o dado passa, uma informação é adicionada a ele para que seja transmitida. No caso do recebimento de informações, cada nível irá retirar aquele pedaço de informação pelo qual é responsável.

- Caso o computador utilize a rede Ethernet, no nível de interface com a rede, encontraremos os seguintes protocolos:**
  - LLC (Controle do Link Lógico);
  - Controle de Acesso ao Meio (MAC);
  - Protocolo físico.

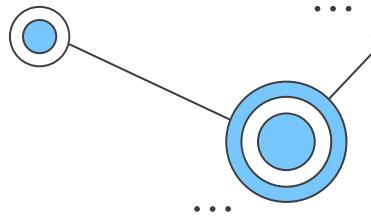


# Protocolo TCP/IP

- Como já mencionamos, as camadas são divididas em funções e classificadas de acordo com as funções mais importantes da rede. Confira a seguir a ordem de cada camada e o nome correspondente às funções executadas por elas:



# Protocolo HTTP

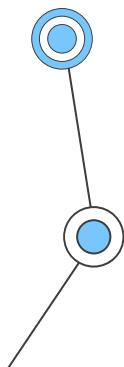


Agora que já conhecemos como funciona o protocolo TCP/IP, vamos falar sobre o protocolo HTTP, que é principal comunicação dos Web Server com o Navegador do cliente.

Em uma comunicação web há sempre dois agentes: O Client e o Server. Para se comunicar precisam de uma conexão e um conjunto comum de regras para poderem se entender. As regras são chamadas de protocolos.

Conceitualmente, ao falar com alguém, é através de um protocolo. Os protocolos na comunicação humana são regras sobre aparência, fala, escuta e compreensão. Trabalham juntos para ajudar as pessoas a se comunicarem com sucesso.

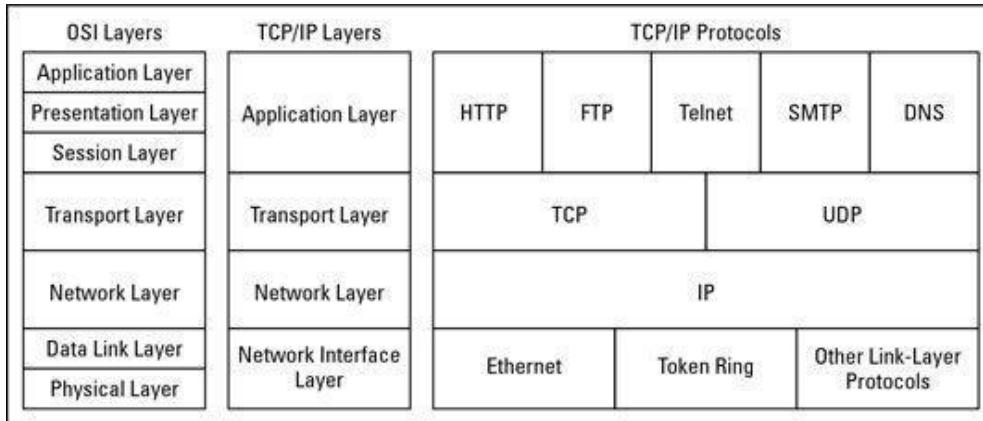
A necessidade de protocolos também se aplica a sistemas de computação. Um protocolo de comunicações é uma descrição formal dos formatos de mensagens digitais e das regras para troca dessas mensagens nos sistemas de computação.



# Protocolo HTTP

O HTTP conhece toda a "gramática", mas não sabe nada sobre como enviar uma mensagem ou abrir uma conexão. É por isso que o HTTP está no topo do TCP/IP.

O HTTP é um protocolo **connectionless**. Significa que o client não se importa se o server está pronto para aceitar uma solicitação e, por outro lado, o server não se importa se o client está pronto para receber a resposta. Mas é necessário uma conexão.



# Versões do IIS

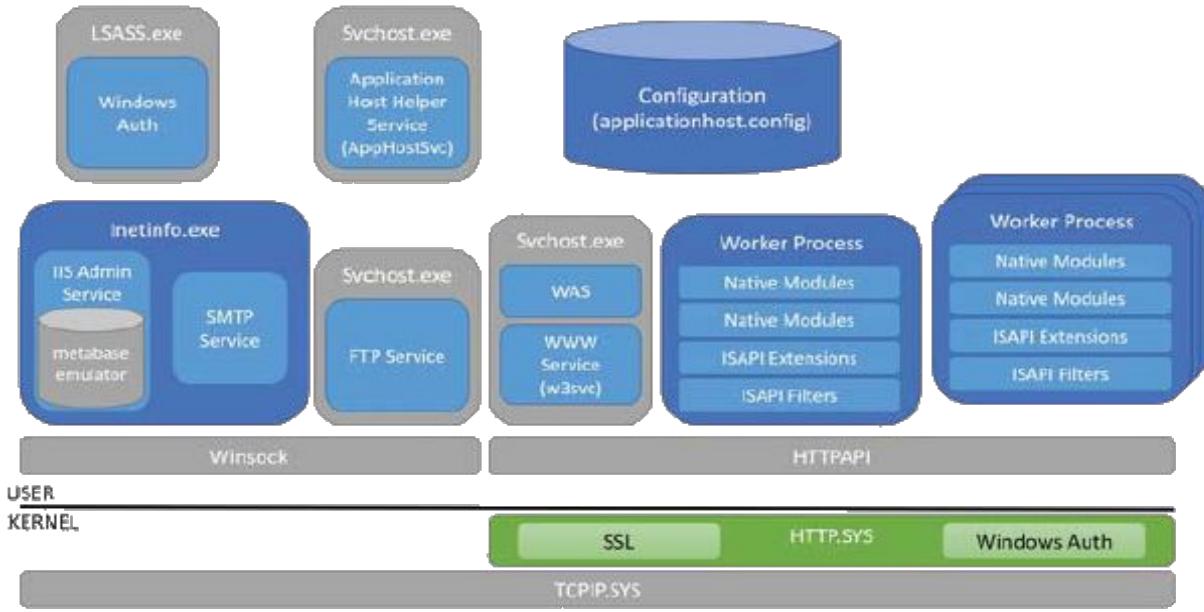
- ☐ Nesta sessão vamos entender o funcionamento do IIS e seus componentes.

A tabela a seguir mostra versões do IIS disponíveis até momento e seus respectivos sistema operacional.

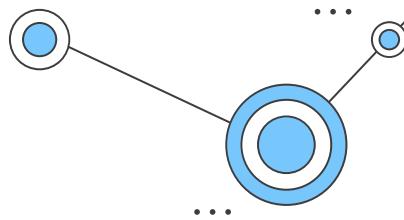
Sistema operacional	Versão do IIS
Windows Server 2003	IIS 6.0
Windows Server 2008	IIS 7.0
Windows Server 2008 R2	IIS 7.5
Windows Server 2012	IIS 8.0
Windows Server 2012 R2	IIS 8.5
Windows Server 2016	IIS 10

# Arquitetura do IIS

- ❑ A figura a seguir exibe os diferentes componentes da arquitetura do IIS:



# Arquitetura do IIS



## □ A arquitetura do IIS tem duas camadas:

- **Kernel Mode:** Tem acesso total a todos os dados de hardware e sistema.
- **User Mode:** Não pode acessar o hardware diretamente e tem acesso limitado aos dados do sistema. Rodar em modo Kernel ou User há desdobramentos até o nível de arquitetura do processador, prioridade de processamento e etc.

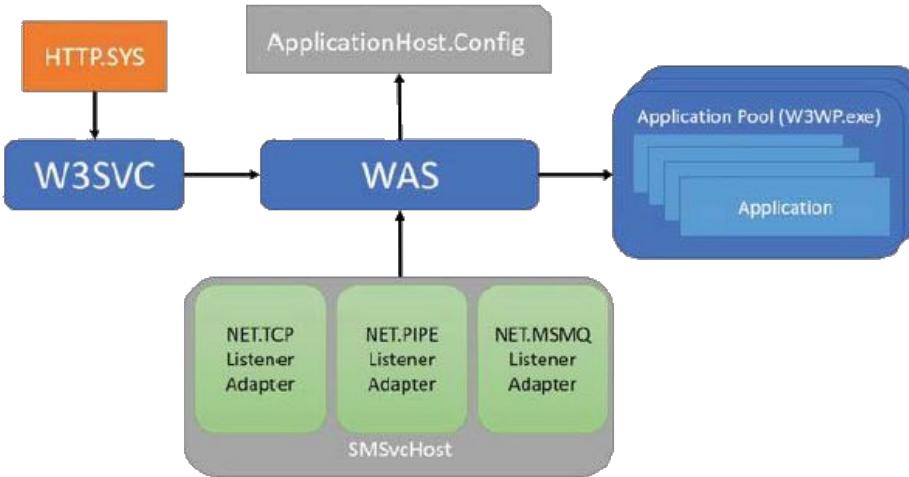


# Componentes do IIS

- Além disso, é importante saber qual é a responsabilidade de cada componente na arquitetura:
  - **HTTP.sys** : É o componente do modo kernel que escuta e recebe as requisições vindas da rede ( `tcp.sys` ), além de hospedar as filas criadas pelo W3SVC. O driver HTTP.sys também responde por caches e grava logs de requisições do IIS, por padrão.
  - **ApplicationHost.config** : Arquivo onde são armazenadas as configurações comuns do IIS, por exemplo, site e Application Pool.
  - **World Wide Web Publishing Service (W3SVC)** : É um serviço hospedado pelo processo `svchost.exe` no modo usuário. Esse componente configura as filas do driver HTTP.sys, de acordo com as configurações descritas no `ApplicationHost .config` . Além disso, é responsável pelos contadores de performance.
  - **Worker Process (w3wp)** : É esse processo que executa o ASP.NET no IIS. É responsável pelo gerenciamento de todas as requests provenientes HTTP.sys. O Worker Process é onde o ASP.NET é executado no IIS.

# Componentes do IIS

- **Windows Process Activation Services (WAS)**: É um serviço hospedado pelo "svchost.exe no modo usuário. Esse componente tem uma grande importância na arquitetura do IIS, pois além de ler as configurações do applicationhost.config , também gerencia o ciclo de vida e saúde do processo denominado worker process ( w3wp.exe ). Ele ainda é responsável por receber as conexões que não são HTTP. A figura mostra o fluxo de acesso ao WAS:



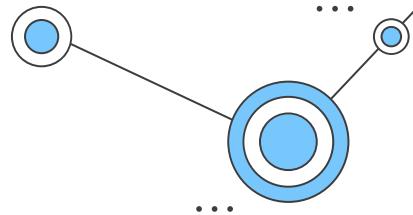
# Componentes do IIS

- **IIS Application Pool :** Um Application Pool pode conter um ou mais Worker Process. Com ele é possível configurar um nível de isolamento entre aplicativos da Web. Por exemplo, se há dois WebSites no mesmo IIS e cada um com seu Application Pool. Os erros em um não afetara o outro.
- **IIS Application Host Helper Service (AppHostSvc) :** É um serviço responsável por habilitar o histórico de configurações do IIS e mapear a identidade do usuário que inicia o application pool. Está hospedado no processo svchost.exe .
- **FTP Service:** Esse serviço está hospedado no processo svchost.exe . A partir da versão IIS 8.0 (antes no inetinfo.exe ), permite que o servidor web forneça funções de FTP (File Transfer Protocol).
- **Inetinfo :** É um componente do modo de usuário que hospeda o arquivo de configuração (metabase) da versão IIS 6 e também serviços que não são web, como: serviço FTP (até versão IIS 7.0) e o serviço SMTP. O inetinfo.exe depende do serviço de administração do IIS para hospedar o metabase.

# Componentes do IIS

- **IISAdminService** : É serviço responsável pela configuração da contabilidade com versão IIS 6.
- **LSASS** : É um processo em modo usuário responsável pela autenticação do Windows. Esse processo tem importância em cenários nos quais clientes não conseguem fazer a autenticação em modo kernel. É importante ressaltar que o driver HTTP.SYS é responsável pela implementação do mecanismo de autenticação integrada do Windows, a partir da versão IIS 7 e posteriores. Esta tarefa foi movida do processo LSASS.EXE no modo de usuário para melhorar o processo de autenticação e reduzir a sobrecarga.
- **Log Trace** : O servidor Web (IIS) possui mecanismos para, em casos de problemas, o time de suporte conseguir rastrear e informar status da requisição, por exemplo, IIS logs e FREB.

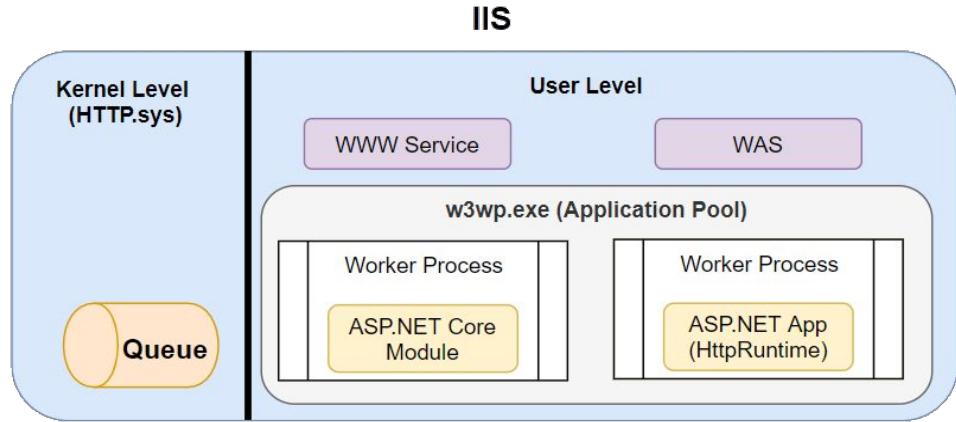
# Fluxo das Requisições

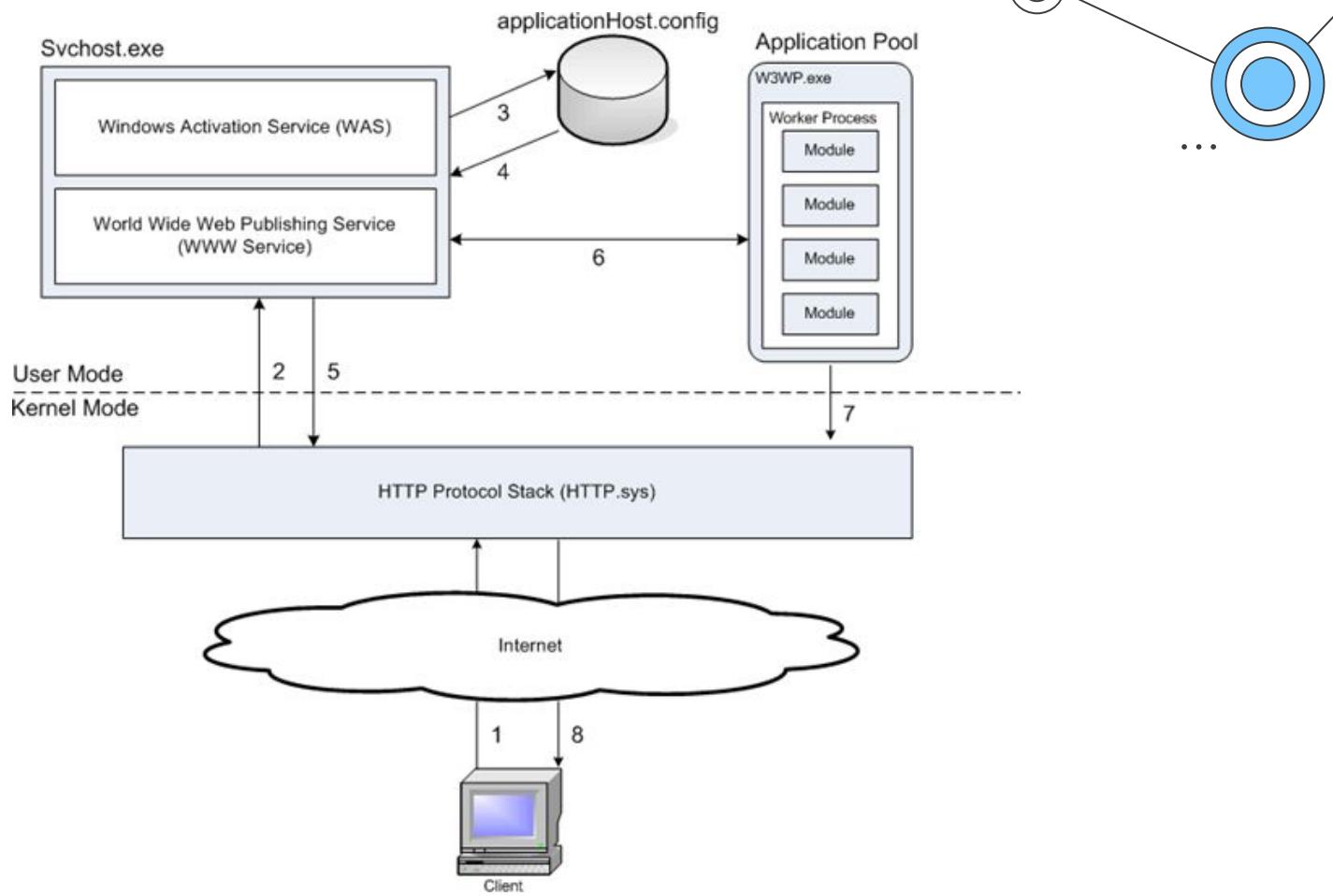


**HTTP.sys** é um listener HTTP. Faz parte do Windows. Ele está na camada **Kernel**. Sua tarefa é interceptar as requests HTTP e enviá-las ao IIS. Depois que a solicitação é processada, o IIS retorna a resposta ao **HTTP.sys** e, por sua vez, retorna a resposta ao client.

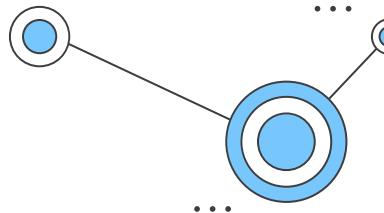
Ao receber um request o **HTTP.sys** põe em uma **Queue** para processamento. Se não houver nenhum **Worker Process** atribuído a fila, o **HTTP.sys** sinaliza o **WWW Service** para criar um.

Quando um **Application Pool** é criado, o IIS cria uma fila de solicitações no **HTTP.sys** e registra-as no **Worker Process** que atende ao **Application Pool**.





# Fluxo das Requisições



**1 -** O Client envia uma requisição. O HTTP.sys intercepta a request HTTP e passa para o IIS. Toda vez que um Application Pool é criado é, também, registrados no HTTP.sys. Que mantém seus IDs. Assim O HTTP.sys identifica o Application Pool da Request. Obrigatoriamente cada aplicativo hospedado no IIS esta atrelado a um Application Pool.

**2 -** O HTTP.sys passa a Request para o WAS.

**3|4 -** O WAS Lê informações do ApplicationHost.config e passa as informações ao WWW Service.

**5 -** O WWW Service Atualiza o HTTP.sys quando há alterações na configuração (Relacionamento entre Application Pool e Request).

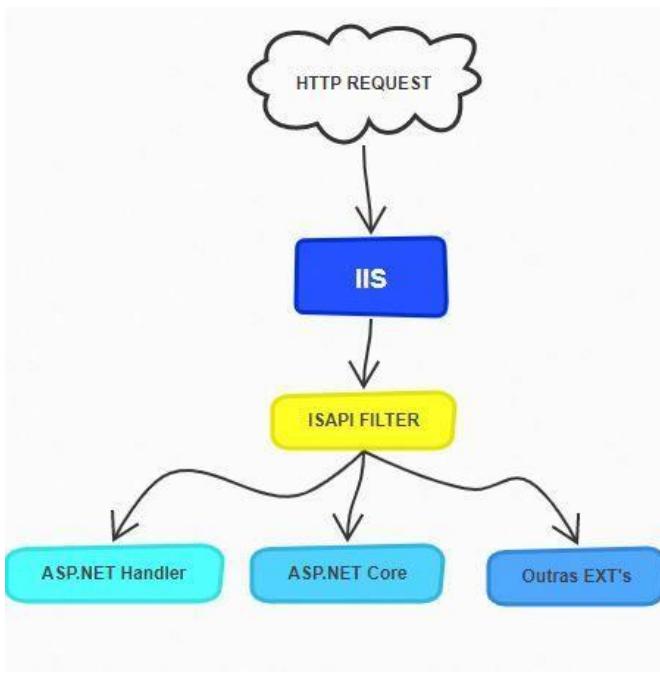
**6 -** O WAS por sua vez, passa a solicitação para o respectivo Application Pool. Se o Application Pool não tiver um Worker Process em execução, o WAS iniciará um.

**7 -** Se o Worker Process estiver ocupado executando outras requests, esta será armazenada na Queue do kernel-mode no HTTP.sys. O Worker Process vai atender a Request assim que tiver uma Thread disponível. O Worker Process lê a URL da Request e o ISAPI Filter carrega a ISAPI extension correta. Caso seja uma aplicação ASP.NET MVC 5 a ISAPI Extension vai iniciar o HttpRuntime. Se for uma aplicação ASP.NET Core, o AspNetCoreModule ou AspNetCoreModuleV2 (.NET Core 2.2).

**8 -** Após isso a request é processada pela Aplicação WEB, retornando essa resposta para o Client através do HTTP.sys.

# Fluxo das Requisições

- Fluxo de Requisição simplificada:



# 05

## Banco de Dados

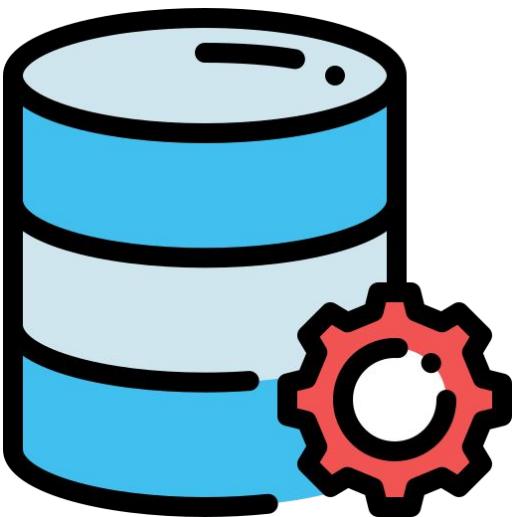
Armazenando dados de  
maneira segura, organizada e  
padronizada.

# O que é Banco de Dados

Existem vários tipos de banco de dados e eles estão presentes na nossa vida há muito tempo, a lista telefônica por exemplo pode ser considerada um banco de dados.

Antigamente as empresas armazenavam informações em arquivos físicos, mas o surgimento e evolução dos computadores possibilitaram o armazenamento de dados de modo digital. Assim os bancos de dados evoluíram e se tornaram o coração de muitos sistemas de informação. A definição de Banco de dados encontrada na internet é essa:

**"Bancos de dados** são coleções de dados interligados entre si e organizados para fornecer informações."



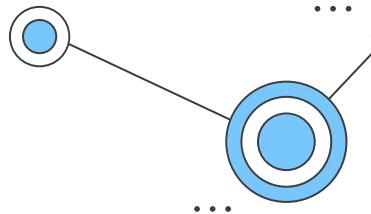
# Dados VS Informação

Muitos consideram **dados e informações** como palavras sinônimas, mas na verdade não são. Para entender o que é um banco de dados é muito importante saber a diferença entre essas duas palavras.

- **Dados** são fatos brutos, em sua forma primária. E muitas vezes os dados podem não fazer sentido sozinhos.
- **Informações** consiste no agrupamento de dados de forma organizada para fazer sentido, gerar conhecimento e facilitar na tomada de decisão, dentro de uma empresa.

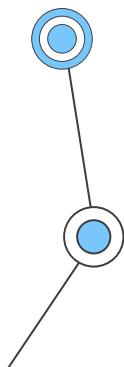
Por exemplo, o número 2001 isoladamente faz algum sentido? Não! Isso é um dado. E se eu dissesse: "Ano do atentado terrorista às torres gêmeas: 2001"? Agora faz sentido! Isso é uma informação. **Um banco de dados é uma estrutura de dados organizada que permite a extração de informações.**

# Proposta de um Banco de Dados



Os bancos de dados foram introduzidos para as aplicações mais diversas, com a proposta de solucionar os seguintes pontos:

- Padronização do acesso:** Ao invés de usarmos softwares e arquivos diferentes para cada tipo de informação, em um banco de dados usamos uma única interface para gerir todas as informações.
- Segurança do acesso:** Em um banco de dados é possível determinar quem acessa o que, e auditar exatamente quem fez o que.
- Integridade das informações:** No banco você pode criar regras que impedem salvamento incorreto ou duplicado de informações.
- Escalabilidade:** Um banco de dados é arquitetado para trabalhar com grandes volumes de dados.
- Trabalho em equipe:** Nos arquivos em geral, só 1 pessoa pode editar por vez. Porém em um banco de dados muitas pessoas podem trabalhar simultaneamente.



# História do SQL Server

SQL Server vem evoluindo ao longo dos anos, tornando-se uma das ferramentas SGBD mais respeitadas e conhecidas do mundo todo.

- Nesta sessão vamos conhecer como esta fantástica ferramenta surgiu, suas versões preliminares e sua evolução nesses longos anos de vida.**
- Em **1988** a Microsoft queria uma solução de Banco de dados em seu portfólio e comprou o direito de vender como um produto (Microsoft SQL Server) uma solução da Sybase e lançou sua primeira versão do SQL Server 1.0, que foi desenvolvida para a plataforma OS/2.
- Durante os **anos 90** a Microsoft iniciou o desenvolvimento de uma versão para a plataforma NT. Enquanto o SQL Server estava sendo desenvolvido a Microsoft decidiu que ele deveria ser uma camada encapsulada sobre o sistema operacional NT, neste período a Microsoft comprou o direito de ler o código do Sybase.
- Em **1992** a Microsoft assumiu a responsabilidade maior sobre o futuro do SQL Server para o NT.

# História do SQL Server

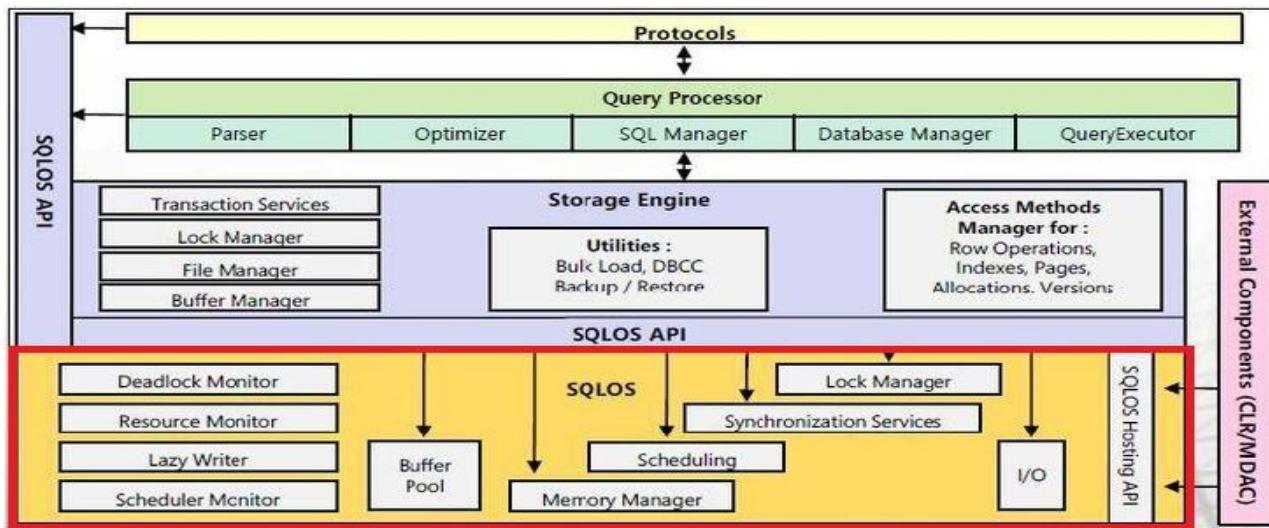
- Em **1993** o Windows NT 3.1 e o SQL Server 4.2 para NT foram lançados. A filosofia da Microsoft em combinar um banco de alta performance com uma interface fácil de usar mostrou-se um sucesso. Microsoft rapidamente tornou-se o segundo mais popular vendedor de softwares de bancos de dados relacionais.
- Em **1994** a Microsoft e a Sybase formalmente encerraram sua parceria.
- Em **1995** a Microsoft lançou a versão 6.0 do SQL Server. Esse lançamento foi uma das maiores rescritas da tecnologia SQL Server. A versão 6.0 aumentou a performance substancialmente provendo mecanismos internos de replicação e administração centralizada.
- Em **1996** a Microsoft lançou a versão 6.5 do SQL Server. Essa versão trouxe melhorias significativas para a tecnologia e disponibilizou diversas novas funcionalidades.

# História do SQL Server

- Em **1997** a Microsoft lançou a versão Enterprise do SQL 6.5.
- Em **1998** a Microsoft lançou a versão 7 do SQL Server o qual foi completamente rescrito.
- Em **2000** a Microsoft lançou o SQL Server 2000. O SQL Server 2000 é o lançamento mais importante do SQL Server até o momento. Essa versão foi construída sobre o framework do SQL Server 7.0. De acordo com o time de desenvolvimento do SQL Server essas mudanças foram desenvolvidas para tornar essa tecnologia mais nova pelos próximos 10 anos.
- Em **2003** O SQL Server 2000 ganha a sua versão em 64-bit podendo acessar maiores quantidades de memória. Infelizmente o time teve de escolher e optou por lançar o SQL Server 2000 apenas para o Itanium.

# História do SQL Server

2005 é lançado o SQL Server 2005 (com o codinome Yukon) com grande integração a plataforma .NET, o time de engenheiros do SQL Server desenvolve o seu próprio conjunto de APIs que é chamado de **SQLOS (SQL Operating System)** e veja ele não é um sistema operacional ele só é um conjunto de APIs que abstrai a camada de sistema operacional, que até então o SQL Server utilizava as APIs (Win32) do próprio Kernel do Windows para manipular os dados no disco. Essa decisão foi tomada para melhorar a forma em que o SQL Server trabalhava.



# História do SQL Server

## □ Um exemplo de como SQLOS mudou a forma de trabalho do SQL Server:

Em uma arquitetura de processadores , temos uma thread que geralmente está vinculada a um processo, e toda vez que o qualquer serviço do Windows precisava executar uma tarefa, é utilizado uma thread que adquire um tempo da CPU e logo após é realizado o processamento sobre a tarefa e qualquer serviço do Windows é **Preemptivo** e isso significa que qualquer execução de uma tarefa demoraria o mesmo tempo que uma consulta do SQL Server.

Então se uma consulta do SQL Server demorava mais que o tempo que do quantum da CPU (tempo da realização da tarefa), ele era obrigado a dar espaço a outra tarefa, pois o tempo que foi reservado para aquela tarefa esgotou. E com o desenvolvimento do **SQLOS** ele conseguiu controlar e priorizar as queries que ele encaminhava para CPU, tornando um sistema **não preemptivo**.

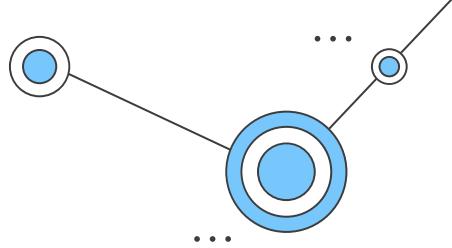
*"No agendamento preemptivo, o sistema operacional pode interromper um processo em execução, com o objetivo de alocar outro processo na CPU. No escalonamento não-preemptivo, quando um processo está em execução, nenhum evento externo pode ocasionar a perda do uso do processador. O processo somente sai do estado de execução, caso termine seu processamento ou execute instruções do próprio código que ocasionem uma mudança para o estado de espera."*

Neste periodo o SQL Server dá mais um passo em direção às grandes plataformas corporativas. A Microsoft exibe alguns grandes casos de sucesso (como a Xerox que consegue realizar até 7.000.000 de transações diárias utilizando o SQL Server 2005 e a Bovespa que é a bolsa de valores do Brasil).

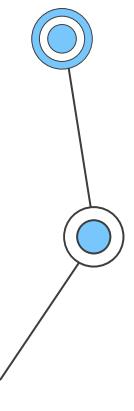
# História do SQL Server

- Lançado em 11 de outubro de **2011**, o **SQL Server 2012** que contaria com novos recursos e aprimoramentos incluem Always On instâncias de cluster SQL Server Fail over e grupos de disponibilidade que fornece um conjunto de opções para melhorar a disponibilidade do banco de dados, novas e modificadas exibições de gerenciamento dinâmico e funções, descoberta de metadados, melhorias de desempenho, tais como índices column store, bem como melhorias e operações de nível de partição e melhorias de segurança, incluindo provisionamento durante configuração, novas permissões e esquema padrão de atribuição para grupos.

# História do SQL Server



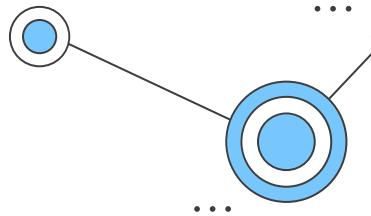
Em **1 de Abril de 2014** o **SQL Server 2014** foi lançado, assim a nova versão fornece uma nova capacidade de memória para tabelas que podem caber inteiramente na memória (também conhecido como Hekaton).



Enquanto pequenas mesas podem ser inteiramente residente na memória em todas as versões do SQL Server, eles também podem residir no disco, por isso o trabalho está envolvido em reservar RAM, escrita despejados páginas para o disco, carregar novas páginas do disco, o bloqueio das páginas na RAM, enquanto eles estão em execução, e muitas outras tarefas.

Para aplicações do SQL Server baseadas em disco, ele também fornece o SSD de Buffers de extensão, o que pode melhorar o desempenho cache entre DRAM e meios de comunicação girando. O SQL Server 2014 também melhora o Always On (HADR), fornece nova recuperação de desastres híbrido e soluções de backup com o Microsoft Azure, que permite aos clientes usar suas habilidades existentes com a no local versão do SQL Server para tirar proveito dos datacenters globais da Microsoft.

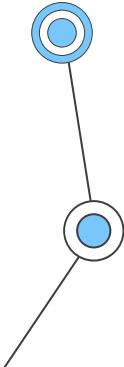
# História do SQL Server



O **SQL Server 2016** oferece funcionalidades inovadoras de missão crítica em termos de escalabilidade, performance e disponibilidade para suas cargas de trabalho de OLTP (Online Transaction Processing ou Processamento de Transações em Tempo Real) e data warehouse mais importantes.

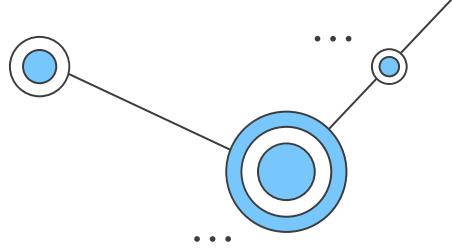
Escale até 12 TB de memória e 640 processadores lógicos com o Windows Server 2016, transações até 30 vezes e consultas 100 vezes mais rápidas com a performance in-memory aprimorada.

Foi introduzido um novo recurso de segurança dos dados em repouso e em movimento com a TDE e a nova tecnologia Always Encrypted.



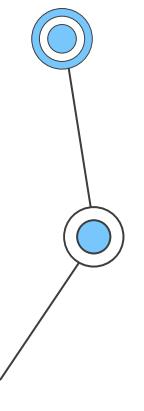
# História do SQL Server

- O **SQL Server 2017** é uma etapa importante para transformar o SQL Server em uma plataforma que oferece opções de linguagens de desenvolvimento, tipos de dados locais ou nuvens e sistemas operacionais, utilizando a capacidade do SQL Server no Linux, em contêineres do Docker baseados em Linux e no Windows. Essa Versão também oferece suporte à implantação de soluções de aprendizado de máquina distribuídas e escalonáveis em várias plataformas, usando várias fontes de dados empresariais, inclusive Hadoop e Teradata.
- O **SQL Server 2019** é a versão atual, lançada em **4 de Novembro de 2019**. introduziu Clusters de Big Data para SQL Server. Ele também fornece melhorias e recursos adicionais para o mecanismo de banco de dados, SQL Server Analysis Services, SQL Server Serviços de **machine learning**, SQL Server em Linux e SQL Server Master Data Services.



# História do SQL Server

O **SQL Server 2019** é instalado em um servidor (este servidor pode ter o sistema operacional Windows ou Linux) ou em um container Docker. Essa é uma das diferenças entre o SQL Server 2019 e o Banco de Dados SQL do Azure que é um serviço na nuvem, ou seja não é instalado em um servidor. As edições disponíveis são:

- 
- **Enterprise:** É a versão completa com recursos datacenter, virtualização ilimitada e BI. Esta versão não é gratuita e o seu licenciamento é feito por núcleo.
  - **Standard:** Permite gerenciamento de bancos de dados (relacionais e BI), oferece suporte a ferramentas de desenvolvimento comuns para rede local e em nuvem – permitindo o gerenciamento eficiente de bancos de dados com mínimos recursos de TI. Esta versão não é gratuita e o seu licenciamento é feito por núcleo.
  - **Web:** Plataforma de dados segura, econômica e altamente escalável para sites. Disponível somente para provedores de serviços. O valor depende do provedor que hospeda o site.
  - **Developer:** Esta edição é idêntica a edição Enterprise, de forma que permite aos desenvolvedores criar, testar e demonstrar de maneira econômica os aplicativos baseados no SQL Server. Esta edição é gratuita.
  - **Express:** Banco de dados de nível básico gratuito ideal para aprendizado, além da criação de aplicativos que utilizam até 10 GB de dados. Esta edição também é gratuita.

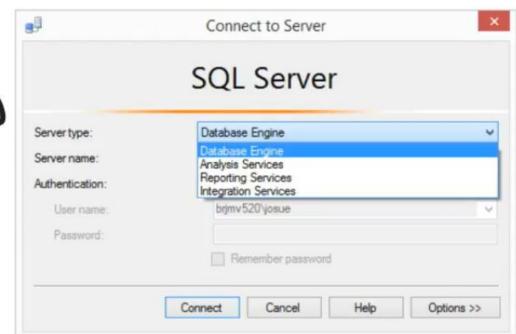
# SQL Server

A maioria pensa que “**SQL Server**” é exclusivamente um banco de dados relacional. Isso era verdade em meados da década de 90, porém, hoje em dia, trata-se de uma solução de softwares que inclui **4 produtos**, como podemos ver na própria janela inicial do “**management studio**”, que é o principal software para administração do SQL:

- Database Engine - SSDE
- Integration Services - SSIS
- Reporting Services - SSRS
- Analysis Services - SSAS



4 softwares



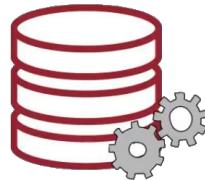
# SQL Server Database Engine

Quando a grande maioria das pessoas diz “**eu uso o SQL Server**”, elas estão dizendo “**eu uso o banco de dados relacional ou o database engine**” da Microsoft. Que hoje em dia, é 1 dos 4 produtos que compõem a solução para banco de dados da Microsoft, que chamamos de SQL Server.

## □ **SQL Server Database Engine**

O **primeiro** produto que compõe a solução SQL Server é o banco de dados relacional, **database engine** ou **SSDE**.

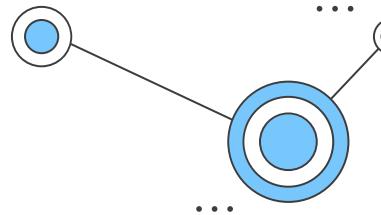
## **DATABASE ENGINE (SSDE)**



Banco de dados Relacional

Guarda informações de transação  
Ex: Estoque, vendas, caixa, etc.

# SQL Server Integration Services



## □ SQL Server Integration Services

O **segundo** produto, é o **SQL Server Integration Services** ou **SSIS**, que é a ferramenta ETL da Microsoft. ETL significa **extract** (extraír a informação de algum lugar), **transform** (transformar a informação se necessário) e **load** (carregar a informação em outro lugar).

O processo de ETL pode ocorrer de qualquer origem para qualquer destino, por exemplo:

- De um banco de dados relacional para outro.
- De um banco de dados para arquivos ou vice-versa.
- De um banco de dados para um data warehouse.
- De um data warehouse para arquivos e vice versa.

## INTEGRATION SERVICES (SSIS)

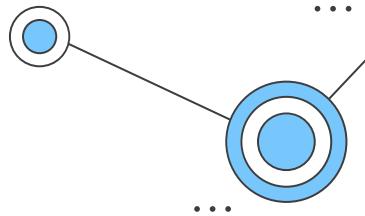
Ferramenta de ETL



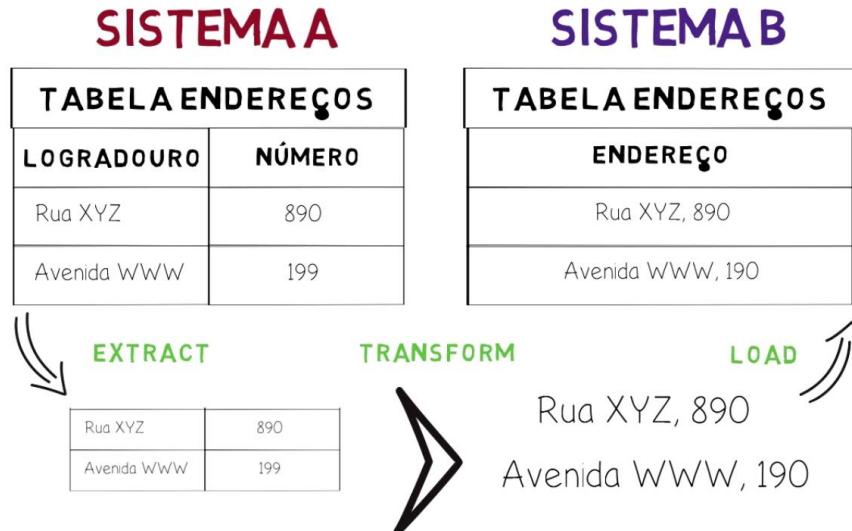
**EXTRACT TRANSFORM LOAD**

Leva dados de um ponto A para um ponto B

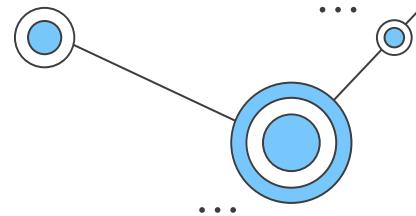
# SQL Server Integration Services



Essa solução tem como objetivo levar dados de um **ponto A**, para um **ponto B**, por exemplo: Em um sistema A você tem os endereços em um campo, e os números em outro, e no sistema B essas informações ficam juntas. Para você migrar as informações do sistema A para o B, você vai extrair do sistema A, transformar a informação, juntando endereço com número, e depois carregar no sistema B.



# SQL Server Reporting Services

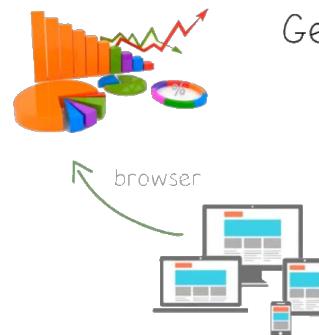


## □ SQL Server Reporting Services

O **terceiro** software que compõe o **SQL Server** é o **Reporting Services** ou **SSRS**, que uma ferramenta para a gestão centralizada de relatórios. O Reporting Services se divide em duas partes:

- A **primeira** é focada no desenvolvimento (criação de relatórios) e para isso você pode usar o C#.
- A **segunda** é focada em disponibilizar os relatórios para os usuários. Para isso, podemos usar o sharepoint da Microsoft, o legal de usar o sharepoint é que como os relatórios ficam todos na web, os seus usuários podem acessá-los de qualquer dispositivo que tenha um browser.

## REPORTING SERVICES (SSRS)



Gestão Centralizada de Relatórios

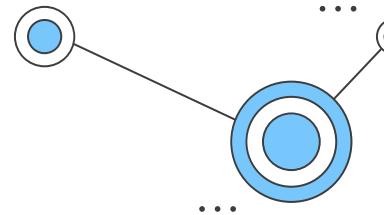
1) Desenvolver



2) Disponibilizar



# SQL Server Analysis Services



## □ SQL Server Analysis Services

O **quarto** e último software é o **SQL Server Analysis Services** ou **SSAS**, que é o data warehouse da Microsoft. Enquanto que o database engine é um banco relacional, o Analysis Services é um banco multidimensional.

## ANALYSIS SERVICES (SSAS)



Data warehouse da Microsoft  
(Banco de dados multidimensional)

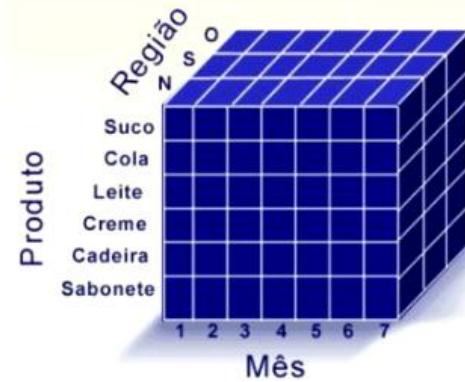
# SQL Server Analysis Services

Enquanto que no mundo relacional ouvimos falar em tabelas, colunas e linhas, no mundo dimensional do **Analysis Services** a organização dos dados ocorre em cubos, dimensões e medida.

SSDE - Relacional

	id	ds_documento	nm	ic_sexo	dt_nascimento
1	12345678900	Ari Tuba		M	1988-01-05
2	12345678901	Dolores Fuertes		F	1978-06-27
3	1234	id	nm	vl	ic_ativo
4	1234	1	Mouse Gammer 25 botões	200.00	1
5	1234	2	Teclado Gammer 350 teclas	300.00	1
6	1234	3	Monitor 32 Pol Full HD	1000.00	1
7	1234	4	RAM DDR4 4GB Powerturbo	500.00	1
8	1234	5	CPU Nasa 10Ghz	2000.00	1
9	1234	6	HD SSD 1TB	2000.00	1
10	1234	7	Notebook Gamer Pro	25000.00	1
					meta
					custo

SSAS - Multidimensional



# Ambiente Corporativo

Vamos dar uma olhada no papel que cada software desempenha no ambiente corporativo.

Na esquerda temos os usuários do Frontend da empresa incluindo transações no **database engine**, por exemplo: vendas, estoques, fluxo de caixa, etc...

A seguir, temos o **Integration Services**, levando as informações do database engine para o **Analysis Services**, que mantem os dados organizados de modo a facilitar o trabalho do Backend da empresa, por exemplo: RH e administrativo.

Por fim, temos o **Reporting Services**, que centraliza relatórios que usam os dados de qualquer um dos bancos de dados que tenhamos em nossa empresa.

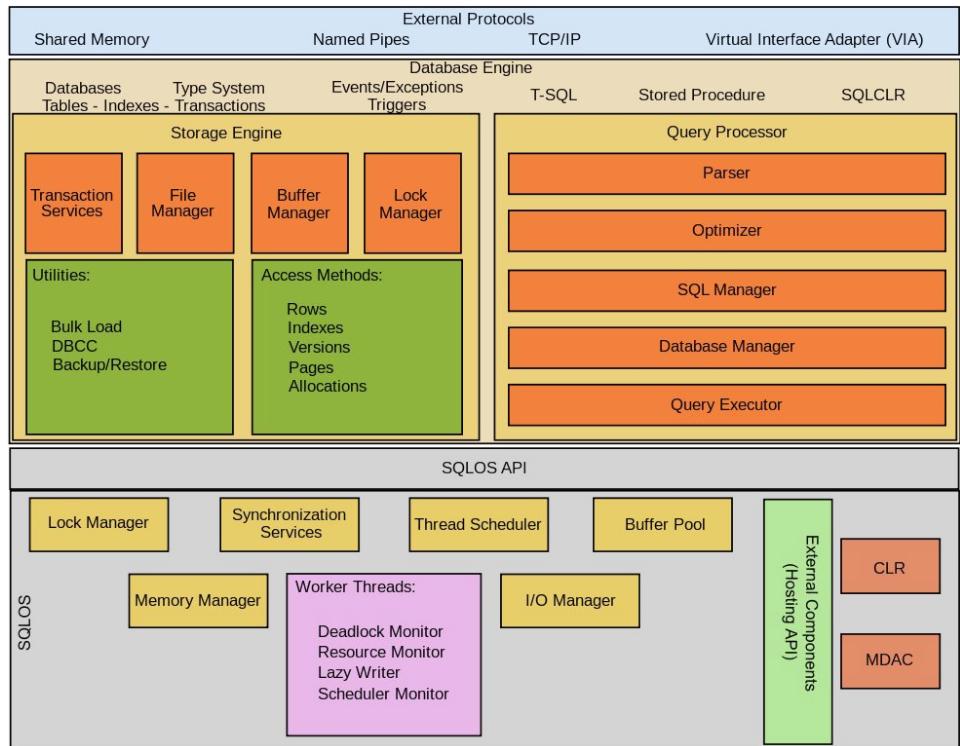
## PAPEL QUE CADA PRODUTO DESEMPENHA NO AMBIENTE DE UMA EMPRESA



# Arquitetura do Database Engine

Microsoft SQL Server é uma arquitetura cliente-servidor. O processo do MS SQL Server começa com o aplicativo cliente enviando uma solicitação. O SQL Server aceita e processa dos dados e retorna as solicitações com os dados processados.

- ❑ O SQL Server consiste em dois componentes principais:
  - Database Engine
  - SQLoS



# Database Engine

O **Database Engine** é o componente principal da arquitetura do SQL Server, destinado ao armazenamento, processamento e proteção de dados. O SQL Server oferece suporte a no máximo 50 instâncias do Database Engine em um único computador. Ele fornece acesso controlado e processamento rápido de transações para atender aos requisitos dos aplicativos de maior consumo de dados nas empresas. Mesmo os objetos de banco de dados, como Store Procedures, Views e Triggers, também são criados e executados por meio do **Database Engine**.

Internamente, consiste em 2 grandes famílias de componentes que é a **engine relational** para processar consultas e um **storage engine** para gerenciar arquivos de banco de dados, páginas, índice, etc.

## Relational Engine

O **Relational Engine** contém os componentes que determinam a melhor maneira de executar uma consulta. O **Relational Engine** também é conhecido como **Query Processor**.

# Query Processor

O **Query Processor** solicita dados do **Storage Engine** com base na consulta de entrada e processa os resultados.

Algumas tarefas do **Storage Engine** incluem processamento de consultas, gerenciamento de memória, gerenciamento de threads e tarefas, gerenciamento de buffer e processamento distribuído de consultas.

Geralmente, ele solicita dados do mecanismo de armazenamento para uma determinada consulta de entrada e processa a saída com base nisso. Existem 3 componentes principais do **Query Processor**. Estes são :

- **Parser (Analizador de Comando)**

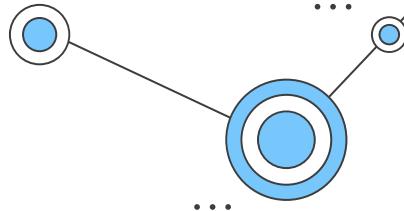
É o primeiro componente do mecanismo relacional a receber os dados da consulta. Ele verifica principalmente a consulta de erros sintáticos e semânticos. E finalmente gera uma árvore de comando.

- **Optimizer**

A principal tarefa do Optimizer é encontrar o plano de execução mais barato, não o melhor e sim um plano com melhor custo-benefício. E a otimização é feita principalmente para comandos DML (SELECT, INSERT, UPDATE, DELETE) e não para todas as consultas. O objetivo final é minimizar o tempo de execução da consulta.

- **Query Executor**

Chama o Método de Acesso. Ele fornece um plano de execução para a lógica de busca de dados necessária para a execução. Depois que os dados são recebidos do Storage Engine, o resultado vai para a camada de protocolo. E, finalmente, os dados são enviados ao usuário final.



# Storage Engine

## □ Storage Engine

O storage engine é responsável pelo armazenamento e recuperação de dados dos sistemas de armazenamento, como discos e SAN. Ele tem os seguintes três componentes principais:

- **Método de acesso**

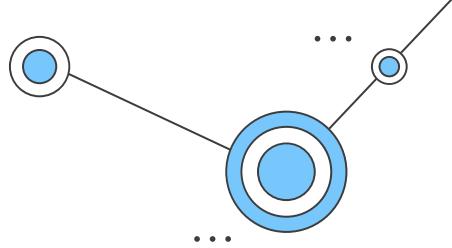
Determina se a consulta é uma instrução selecionada ou não selecionada. E então invoca o buffer e o gerenciador de transferência de acordo.

- **Gerenciador de buffer**

Gerencia as funções básicas do cache do plano de execução, análise de dados e página suja.

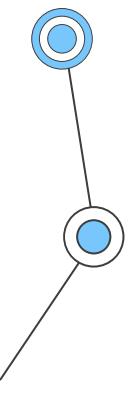
- **Gerenciador de transações**

Gerencia transações não selecionadas com a ajuda de gerenciadores de log e bloqueio. Além disso, promove a implementação de logs Write Ahead e gravadores Lazy.



# SQL Server OS

O **Sistema Operacional do SQL Server (SQLOS)** é uma camada de aplicativo separada no nível mais baixo do **Database Engine** do SQL Server em que o SQL Server e o **SQL Reporting Services** são executados na parte superior. Ele foi introduzido no SQL Server 2005. Ele está sob o **Query Processor** e **Storage Engine**. Na verdade, ele fornece serviços de sistema operacional, como gerenciamento de memória e controle de I/O (leitura e escrita), incluindo outros serviços que incluem controle de locks, tratamento de exceções, agendamentos de tarefas e serviços de sincronização.



## □ Na verdade, ele executa as seguintes funções críticas para o SQL Server:

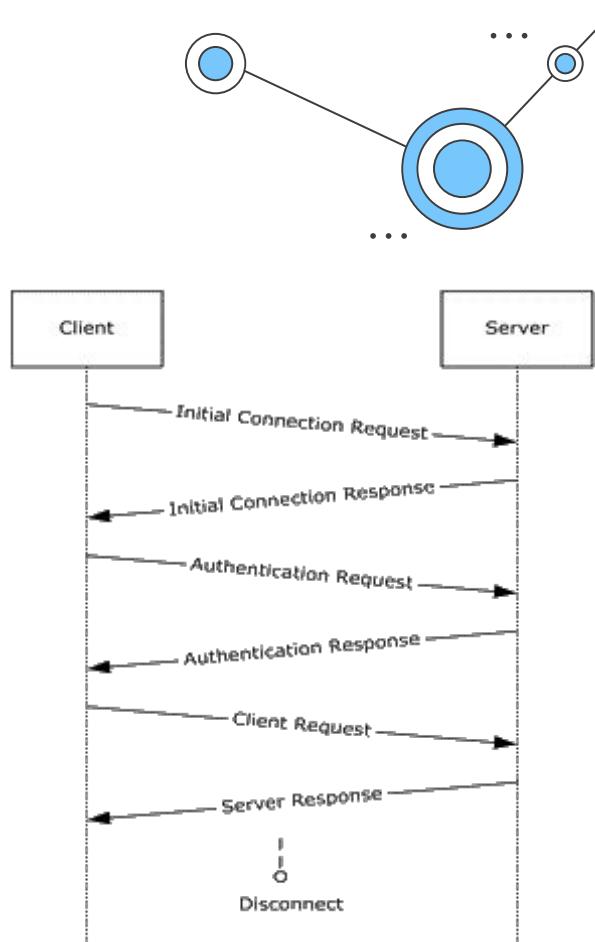
- Conclusão do agendador e do IO: o SQLOS é responsável por sinalizar as threads quando o IO é concluído.
- SQLOS é responsável por gerenciar sincronizações de thread.
- Estrutura de tratamento de exceções.
- Detecção e gerenciamento de deadlock.
- O SQLOS pode controlar a quantidade de memória que um componente do SQL Server está consumindo.
- Serviços de hospedagem para componentes externos, como CLR e MDAC.

# Protocolos Externos

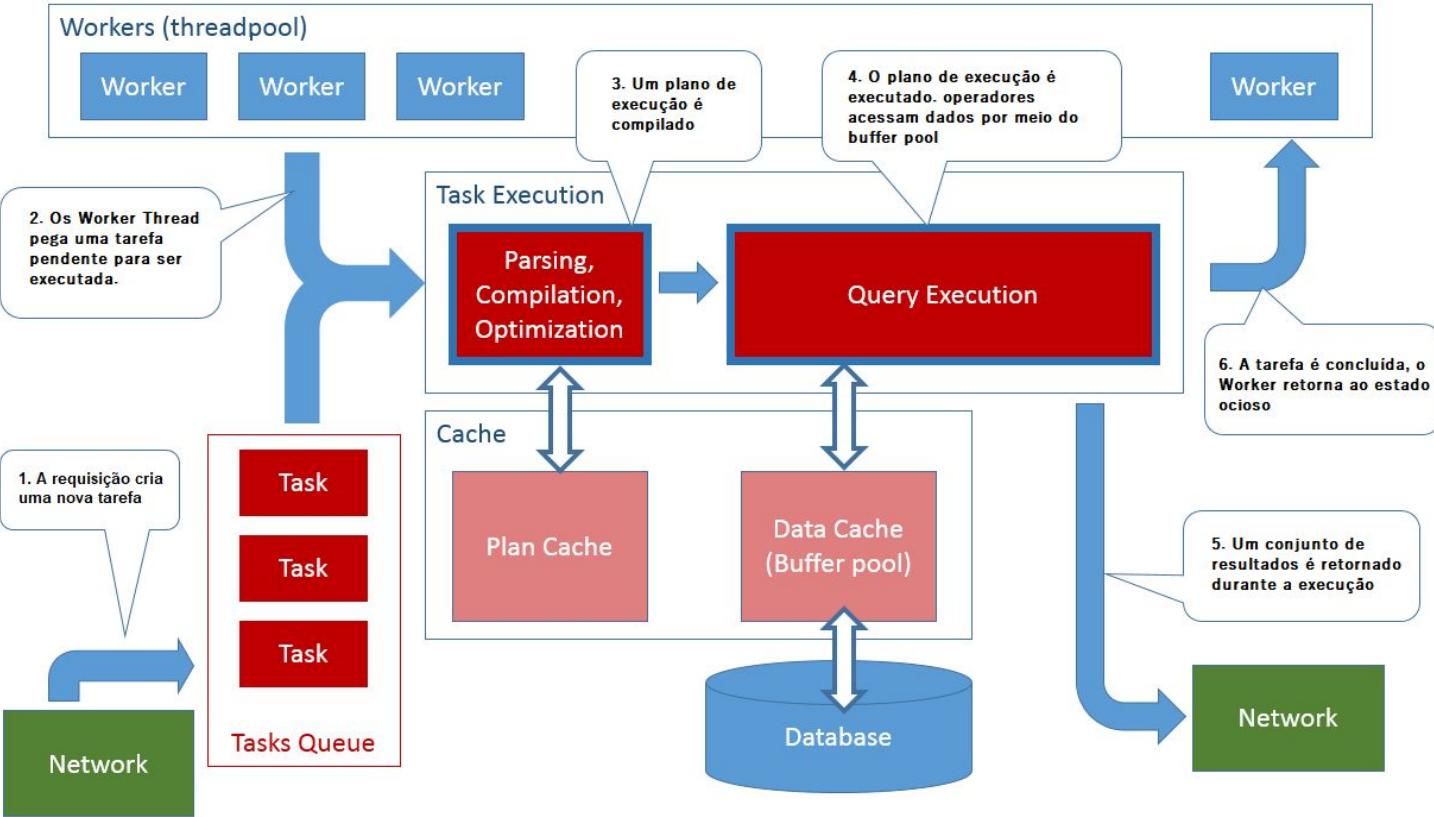
A camada de protocolo implementa a interface externa para o SQL Server. Todas as operações que podem ser chamadas no SQL Server são comunicadas a ele por meio de um formato definido pela Microsoft, chamado **TDS (Tabular Data Stream)**.

Esta interface pode usar uma das várias implementações do lado cliente do protocolo: a CLR gerencia SqlClient, OleDB, ODBC, JDBC, drivers PHP para SQL Server ou a implementação aberta FreeTDS. A essência principal, é que, quando o aplicativo pede ao banco de dados para fazer qualquer coisa, este envie um pedido através do protocolo TDS.

A conexão é estabelecida diretamente vinculado à sessão de nível de transporte. O Servidor persiste até que a conexão seja encerrada (por exemplo, quando um soquete TCP é fechado). Além disso, o TDS não faz nenhuma suposição sobre o protocolo de transporte usado, mas assume que o protocolo de transporte oferece suporte a entrega confiável e ordenada os dados.



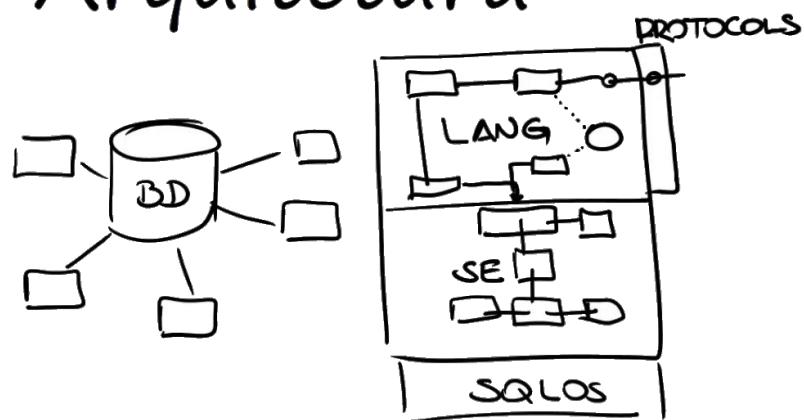
# Fluxo de uma Query



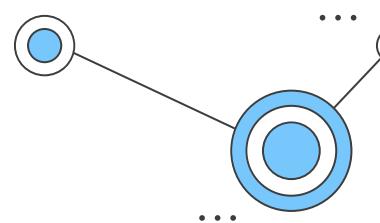
# Fluxo de uma Query

A porta de entrada para as requisições é feita através do **SQL Protocols**, que recebe a requisição do cliente e empacota as informações para dentro do serviço do SQL, para ser mais especificamente no componente de **Linguagem (Query Executor)**, a primeira ação tomada dentro do SQL Server é o roteamento, se a requisição é uma consulta SQL ela passa por um processo de compilação para em seguida fazer a sua execução acessando os métodos do **storage engine**, porém se a requisição for uma store procedure ela terá um atalho, neste caso é possível fazer uma consulta ao procedure cache e verificar se já existe um plano de execução e se existir a execução é feita imediatamente, pulando a etapa de compilação.

## Arquitetura



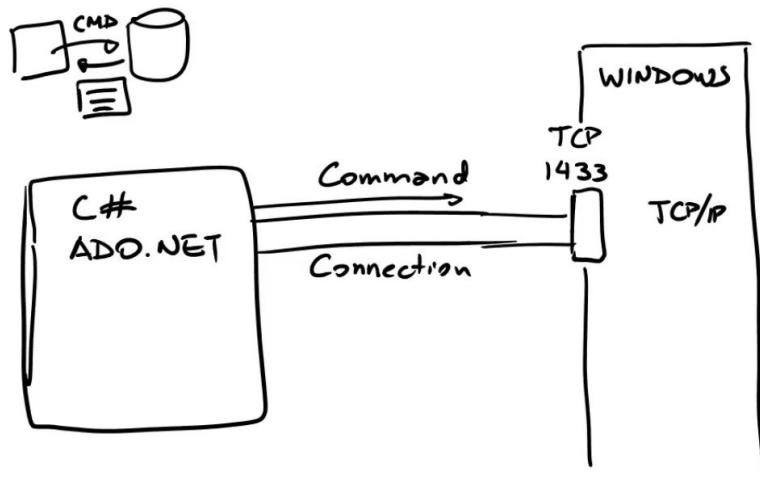
# Conexão



Vamos acompanhar o trajeto completo de uma consulta para o SQL Server, primeiramente temos o cliente que fala com o banco de dados ele envia um comando para o banco de dados e espera receber uma tabela com as informações que ele pediu.

Digamos que essa solicitação do cliente foi escrita em C# utilizando o ADO.NET, inicialmente é aberto uma conexão com o banco de dados e nesta conexão (Connection) é enviado o comando (SQLCommand).

Do lado do SQL Server temos uma porta TCP 1433, essa porta recebe a informação e coloca para dentro do Sistema Operacional Windows.

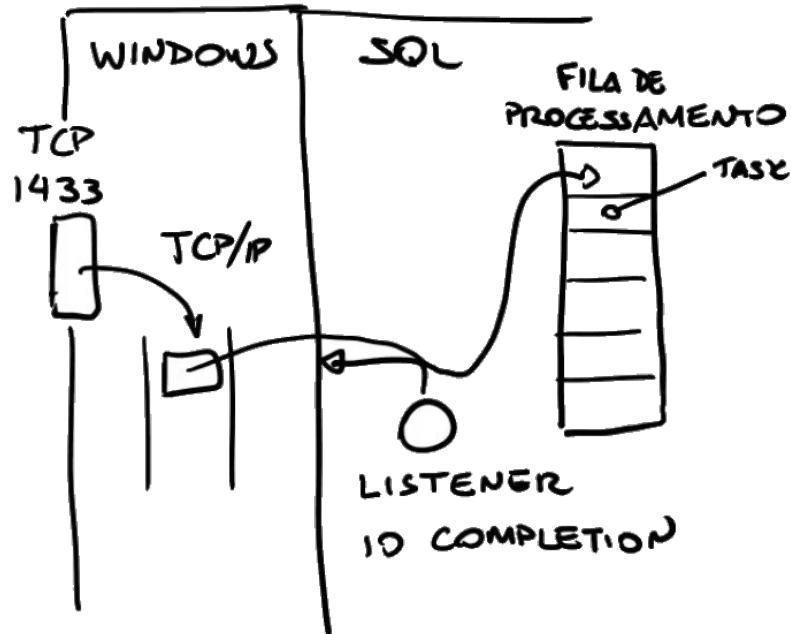


# Fila de Processamento

O Windows possui drivers e um deles é o TCP/IP e o TCP/IP tem a sua própria fila.

Do lado SQL Server existe uma Thread chamada **Listener** mais precisamente **I/O Completion** que completa as requisições assíncrona, neste caso ela vai escutar a porta TCP/IP e vai tirar essa requisição e vai colocar para dentro do SQL Server em uma fila de processamento, ou seja a requisição sai pilha TCP e vai para fila de processamento interna do SQL Server e agora o processamento começa.

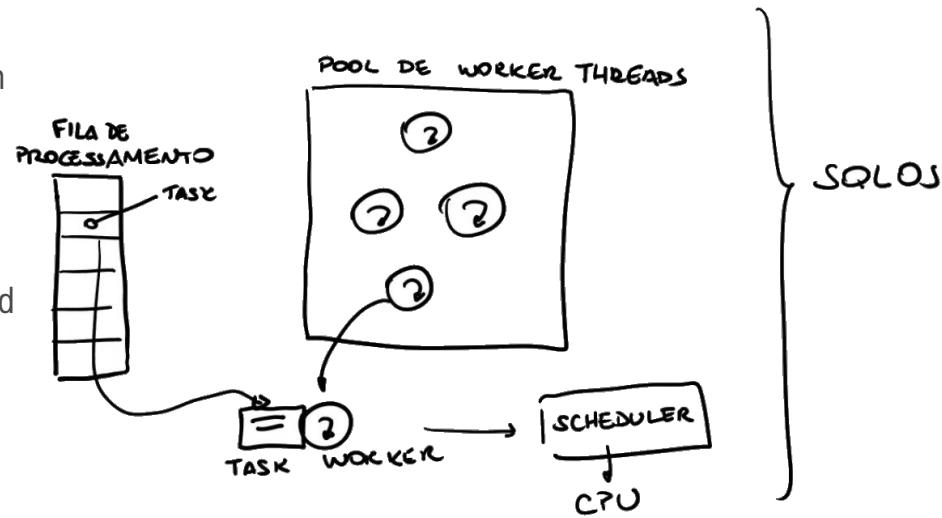
Essa fila de processamento é composta por uma série de tarefas (Task).



# SQL OS

Uma vez a **Task** dentro do SQL Server ela pode ser agendada pelo serviço de **scheduler**, isso significa que o scheduler tem uma série de threads (Worker Threads) disponíveis e todas elas vivem dentro de um **pool**, ou seja, elas estão disponíveis para executar uma tarefa e quando tiver uma tarefa na fila, o scheduler pega uma das Worker Threads e prepara para executar a tarefa designada, desta forma estamos anexando uma Task com uma Worker Thread e neste momento o scheduler enxerga essa Worker Thread como uma thread ativa ou **RUNNABLE**.

Isso significa que ele pode pegar essa worker e agendar em uma determinada CPU do Sistema Operacional essa é uma camada fundamental e faz parte do **SQL OS**.

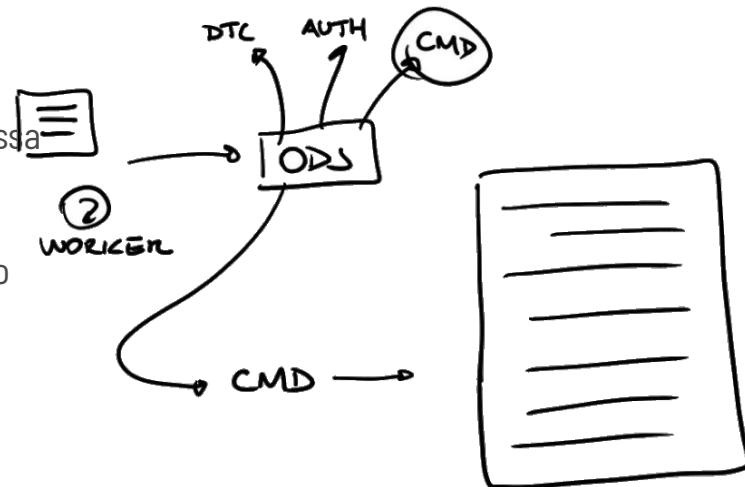


# ODS (Open Data Service)

Pronto, Agora estamos dentro do SQL Server a Task que veio do usuário com uma Worker Thread responsável pela execução e a primeira ação que vai fazer é pegar a Task e jogar para um roteador.

Esse roteador se chama **ODS (Open Data Service)** ele vai abrir essa mensagem que é a tarefa e determinar qual o tipo dela e rotear para a rotina adequada, essa tarefa pode ser do tipo DTC, AUTH (Autenticação) ou CMD (Comando) e este comando pode ser tanto uma consulta AdHoc como também uma store procedure.

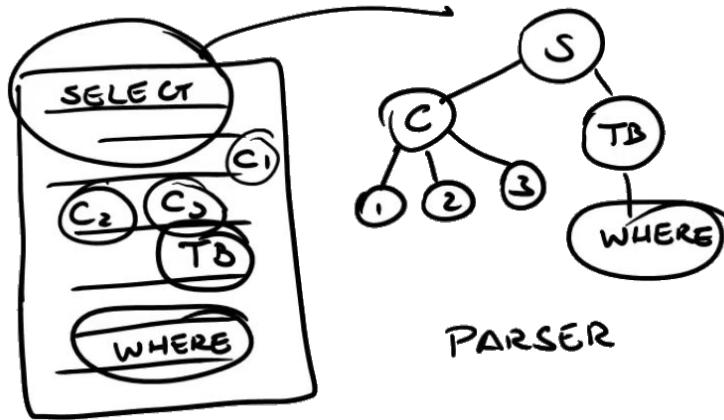
Vamos supor que esta tarefa seja a mais comum que é a de Comando, então o ODS fez o roteamento e enviou a tarefa para a rotina de processamento de comando essa rotina é responsável por concatenar os pacotes de rede 4k e 8k e gerar o **input buffer** que é composto por vários caracteres textos.



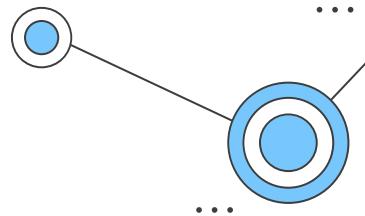
# Parser

O Processamento do input buffer é feito em partes, primeiro ele identifica o tipo de comando, se for uma consulta SELECT é colocado em uma representação de arvore, depois é verificado quais são as TABELAS, COLUNAS E FILTROS utilizados e também é colocado nesta representação de arvore e essa é a fase do **PARSER**.

Essa etapa é conhecido como reconhecimento da gramática, onde **Query Processor** transforma um bloco de texto em um formato interno, e essa estrutura de arvore gramatical ela certamente define qual informação será pesquisada entretanto existe várias formas para acessar este dado.

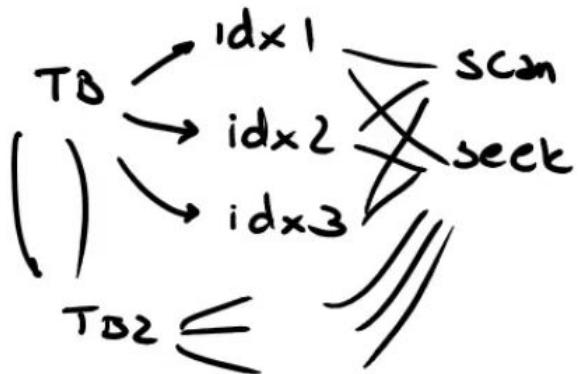


# Parser



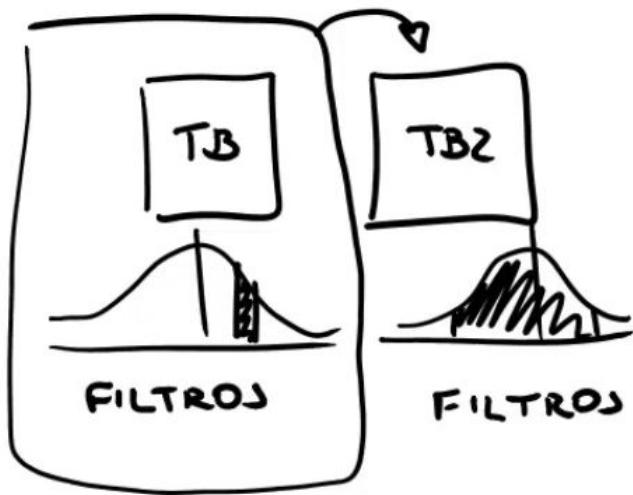
Um exemplo das formas de acesso aos dados é buscando pelo índice e uma tabela possui vários índices: (Idx1, Idx2, Idx3) e em cada índice pode ser realizado diversos tipos de operação Ex: scan, seek.

Esse caso houver outras tabelas na consulta com vários índices temos mais formas de acesso e com essas outras tabelas implementamos mais complexidade pois agora o Query Processor precisa verificar a ordem de acesso e para resolver essa questão o Query Processor leva em consideração a distribuição de dados nas tabelas e todas as tabelas possuem informações estatísticas.



# Parser

E essa estatística é representada como um histograma de dados que é um valor aproximado da quantidade de registros que possui na tabela, já os filtros usados na cláusula WHERE são usados como fator determinante, porque com eles é restringido a quantidade de informações que é consultada. O Optimizer começa sempre com a menor tabela ou com o filtro mais sensitivo o objetivo é minimizar o custo da operação, ou seja diminuir o numero de acesso a memória, disco etc.



# Plano de Execução

Agora que SQL Server passou pela etapa de construir uma árvore gramatical, a próxima etapa é de gerar **plano de execução**. A ideia do plano de execução é transformar **qual** informação necessária que foi obtido da consulta para **como** buscar essa informação no disco.

## ❑ Exemplo:

1. É iniciado pela tabela 1 e pega os registros.
2. Faz a consulta na tabela 2, faz o cruzamento dos dados e retorna o registro combinado.
3. Temos o resultado Final.

## ❑ Algo Parecido como:

Um **looping** utilizando **for** onde **i** começa com **0** percorrendo todos os registros e aí é adicionado a ordem da consulta, primeiro é feito a consulta na **tabela 1** e com base neste registro é feito a consulta na **tabela 2**.

Execution Plan

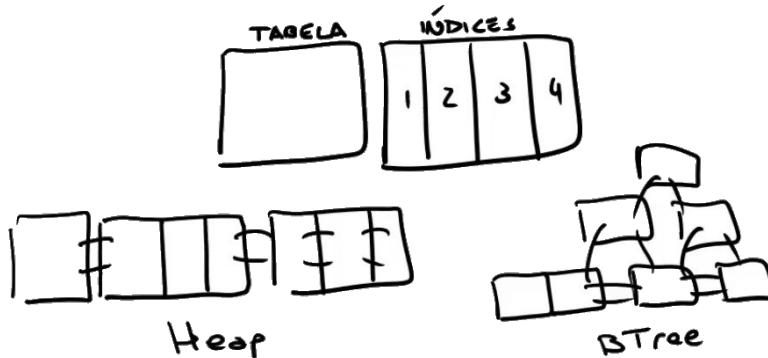


```
for (i=0; i < len; i++) {  
    reg = TB[i];  
    reg2 = TB2[reg];  
}
```

# Estrutura das Páginas

Agora temos 2 formas de ver essa representação a primeira seria a forma tradicional um plano de execução representada de forma **gráfica** onde quadrado corresponde a um operador. Uma outra forma seria com **Reggaeton** onde ele gera esse mesmo plano de execução só que como uma **dll**, ou seja, ele é compilado.

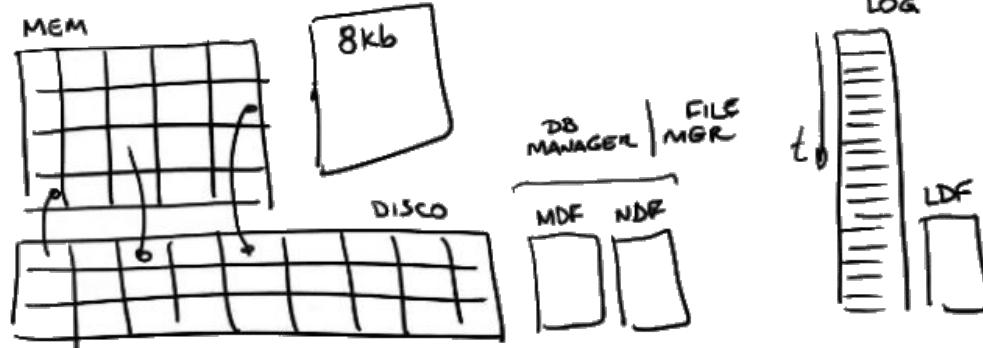
E depois do plano de execução conseguimos rodar esse programa, seja ele no formato de dll ou composto por vários operadores, no fundo a execução observa os componentes lógicos como as tabelas e índices, porém por debaixo dos panos uma tabela é composta por várias páginas de dados elas são interligadas como uma **lista duplamente ligada** e os índices também possuem a mesma estrutura, porém os índices possuem uma estrutura superior de árvore e essa estrutura é chamada de **B Tree (Arvore Baçanceada)**, enquanto uma tabela sem índice cluster é chamada de **heap**.



# Página de Dados

E tanto a estrutura **heap** e **btree** é composto por blocos de 8 KB de tamanho fixo esse bloco pode estar em memória ou em disco essas páginas de dados são carregadas do disco para a memória ou descarregadas da memória para o disco e essa é a principal função do banco de dados fazer o cache de dados e toda essa memória é armazenada no **buffer pool**, organizada pelo **buffer manager** e todas essas informações estão localizadas fisicamente em arquivos, o arquivo **.mdf** é o primário e os **.ndf** que são os secundários, esses arquivos são gerenciados pelo **database manager** que coordena a parte de banco de dados e individualmente existem os **files manager**, além disso existem mais um tipo de arquivos que são os **arquivos de log** e o **transaction manager** é responsável por gerenciar **todas as escritas** no banco de dados.

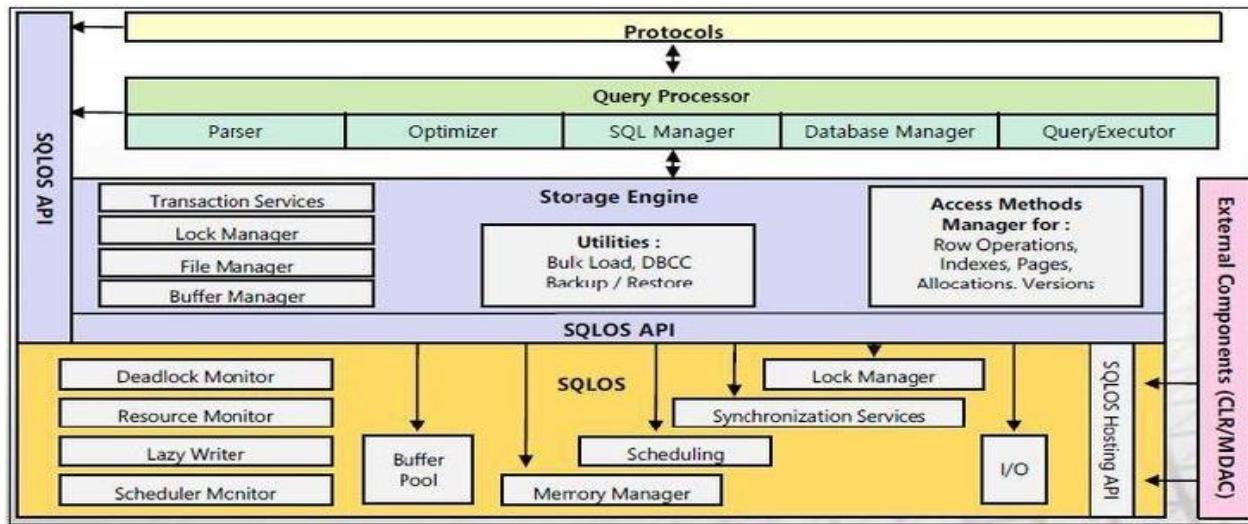
As escritas não é feita só no buffer pool todas as operações são logadas em um log sequencial e incremental ao longo do tempo e essas informações garantem a consistência dos dados.



# Conclusão

O SQL Server possui uma arquitetura bem projetada, primeiro ele possui um componente focado em **linguagem (Query Processor ou Relational Engine)** que recebe a informação em modo texto e compila em um executável, assim como temos a parte de armazenamento que é o **storage engine** e por fim a camada de **protocolo** que recebe a informação e repassa para o **SQL OS**.

- Fluxo: **Protocols** -> **SQLOS** -> **Query Executor** -> **Storage Engine**.



# Obrigado!

Alguma Pergunta?

- Você pode me encontrar em:
  - ❖ willianbrito05@gmail.com
  - ❖ www.linkedin.com/in/willian-ferreira-brito
  - ❖ github.com/willian-brito



# Referências

- <https://acervolima.com/introducao-ao-sql-server-arquitetura/>
- <https://www.youtube.com/watch?v=9lleeVTTC5s>
- <https://www.sqlservertutorial.net/getting-started/what-is-sql-server/>
- [https://www.youtube.com/watch?v=WZf\\_F0vHefg&t=657s](https://www.youtube.com/watch?v=WZf_F0vHefg&t=657s)
- <https://www.youtube.com/watch?v=9FDzw65KSIq&t=1152s>
- <https://www.brunobrito.net.br/anatomia-de-uma-aplicacao-asp-net-iis-parte-i/>
- <http://www.linhadecodigo.com.br/artigo/120/arquitetura-net-msil-clr-cts-cls-blc-jit.aspx>
- <https://www.clubedohardware.com.br/artigos/redes/como-o-protocolo-tcp-ip-funciona-parte-1-r34823/>