# Básicos

## Memória

* **RAM**: A ram é dividida em vários espaços de 1byte cada espaço possui um endereço que vai de 0 até n. Endereços são escritos em hexadecimal com o prefixo 0x (ex: 0x3D4)
* A CPU pode solicitar que a memória leia ou escreva em um endereço x
* A CPU não guarda nenhum endereço; ela recebe de um programa a instrução para escrever ou ler em um endereço.
* Não existe nenhum controle de acesso nesse nível. Um programa pode teoricamente ler/escrever em qualquer endereço.

### Controle de acesso à memória

* O controle de acesso à memória é feito pelo Sistema Operacional; um programa não acessa a CPU/memória diretamente. Ele precisa passar pelo SO antes de acessar a memória. Isso é feito da seguinte maneira:
  + O SO divide a memória em unidades chamadas **páginas** ou **endereços virtuais**. Geralmente, cada página tem 4 KB (4096 bytes), mas é importante lembrar que em alguns sistemas, as páginas podem ter outros tamanhos, como 2 KB, 8 KB, ou até mesmo 1 MB ou 2 MB em casos de páginas grandes (huge pages).
  + Uma página pode pertencer a um programa, e quando isso ocorre, essa página só é visível para esse programa, e não para outros (a menos que seja uma página compartilhada). Cada página possui permissões de escrita, leitura, e execução.
  + Se um programa tentar acessar uma página da qual ele não tem acesso, o SO gera uma interrupção para esse programa através da CPU, geralmente resultando em um erro de “segmentation fault” ou “access violation”.

### **Cache**:

* É uma memória de baixa capacidade, mas extremamente rápida, localizada na CPU, para que ela não precise acessar a RAM para operações frequentes e simples.
* Os dados que estão no cache também estão na RAM. O cache é dividido em unidades de 64 bytes chamadas **cache lines**.

### Memória virtual:

* Uso de espaço em disco para simular memória adicional. Páginas de memória que não estão atualmente em uso pela RAM são armazenadas no disco.
* Quando essas páginas são solicitadas por um programa, o SO deve liberar espaço na RAM e mover as páginas do disco para a RAM. Esse processo é conhecido como **swap** e pode tornar o acesso mais lento.

### Heap/dynamic memory

* É um bloco de memória não estruturado que é alocado manualmente por um programa. O tamanho do bloco é definido pelo programa e é geralmente usado para alocações dinâmicas que não são conhecidas em tempo de compilação. A memória no heap é desalocada explicitamente pelo programador ou quando o programa termina.
  + Em **C**, existe a função malloc(x), que recebe como parâmetro a quantidade de memória que deve ser alocada, retornando um ponteiro para a memória alocada ou NULL se falhou em alocar. Esse ponteiro deve ser usado para desalocar a memória através da função free().
  + Em **Java**, a memória heap é alocada usando new ou criando um array. O tamanho da memória alocada é gerenciado pela JVM, normalmente correspondendo ao tamanho do objeto. A memória é automaticamente desalocada pelo Garbage Collector (GC)

### **Memory leak:**

* Ocorre quando um programa perde a referência para a memória alocada sem que ela seja liberada, fazendo com que a memória fique alocada até que o programa termine.
* Em **C**, isso acontece quando free() não é chamado. Em **Java**, ocorre quando o GC não consegue remover um objeto não referenciado por algum motivo (embora o GC seja projetado para mitigar vazamentos de memória).

### Stack

* É uma área de memória usada para gerenciar o fluxo de execução de um programa (stack de chamadas) e guardar dados temporários como variáveis locais, argumentos de funções, etc.
* A stack é uma estrutura de dados **LIFO** (Last In, First Out), permitindo rápida leitura e escrita, mas possui um tamanho máximo. Se todo o espaço for utilizado, ocorre um “stack overflow”.
* A alocação e desalocação de memória na stack são gerenciadas automaticamente pelo compilador e a linguagem de programação.

#### ****Componentes da Stack****:

* **Stack Pointer**: Um ponteiro que aponta para o topo da stack, ou seja, os dados da função atualmente em execução.
* **Stack Frame**: estrutura que contém o endereço de retorno da função que chamou a função atual, os parâmetros passados e suas variáveis locais.
* Quando uma função é chamada, um stack frame é adicionado à stack, e o stack pointer é atualizado para apontar para esse novo frame, indicando que aquela função está sendo executada.
* Quando a função termina, o stack pointer volta a apontar para o stack frame anterior, e o stack frame da função que terminou é removido da stack.
* Criar as instruções necessárias para esse processo é responsabilidade do compilador. Em alguns casos, o compilador otimiza o código removendo variáveis não utilizadas para reduzir o tamanho do stack frame e melhorar a eficiência.

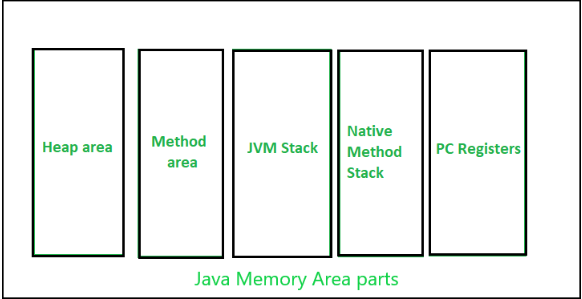


# JVM e GC

* A maior parte do gerenciamento de memória em Java é feita pela JVM, com o GC garantindo que a memória alocada e não mais utilizada seja liberada.
* Problemas como a destruição de objetos são tratados pelo GC, mas ele não gerencia todo o processo. Algumas responsabilidades ainda são do desenvolvedor, como evitar referências que impedem que o GC remova objetos.

## Estrutura da memória da JVM

* A JVM possui sua própria estrutura de memória. Algumas partes são criadas pela própria JVM e só são destruídas quando a JVM para de executar, enquanto outras são criadas por threads e só são destruídas quando uma thread é finalizada.
* A memória da JVM é memória virtual
* A parte da JVM que contém a estrutura de memória é a Runtime Data Area



### Heap

* É a área de memória onde os objetos (instâncias de classes e arrays) são armazenados. É alocada durante o startup da JVM, e seu tamanho pode ser fixo ou dinâmico, dependendo da configuração.
* Existe um heap por processo de JVM.
* Quando um objeto é criado, ele aloca espaço no heap, e uma referência para ele é criada na stack.
* O GC atua principalmente no heap, sendo responsável por liberar objetos que não são mais referenciados.

#### String Constant Pool ou interned strings

* O **String Constant Pool** é uma área da memória, usada para armazenar strings e permitir sua reutilização
* Até o Java 7 fica na Method Area do heap, após a remoção da PermGem fica em uma área normal do heap
* **Reuso de Strings:** Quando uma string é criada, a JVM verifica se essa string já existe no pool. Se existir, ela reutiliza a referência existente; caso contrário, uma nova string é criada e adicionada ao pool.
* Método intern()**:** O método intern() tenta adicionar uma string ao String Pool. Se a string já existe, ele retorna sua referência.
* GC ocorre normalmente

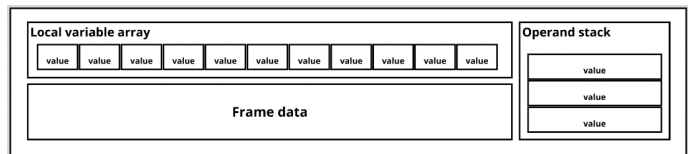
### Metaspace

* No java 8 substituiu uma area chamada de method class area ou permanent generation
* É uma area de memória nativa. Memória nativa é memória gerenciada e provida pelo SO e é compartilhado entre todas as threads
* O metaspace é chamado assim por guardar metadados de classes, os metadados são: class files, estrutura e métodos, constantes, anotações e otimizações
  + São os metadados do class loader usados para instanciar objetos
* Variáveis de classe static e string pool ficam no heap

### JVM Stack

* Armazena tipos primitivos e ponteiros para objetos no heap em frames. O tamanho da stack pode ser fixo ou dinâmico, dependendo da implementação de JVM usada
* Cada thread possui sua própria stack.
* Para cada chamada de método um frame novo é criado, esse frame guarda os dados do método como retorno, variáveis, etc.
* Se o tamanho da stack não for fixo e não puder alocar mais memória, pode lançar um **OutOfMemoryError**. Se o tamanho for fixo e toda a memória for utilizada, será lançado um **StackOverflowError**.
* A alocação e desalocação de memória é mais rápida do que no heap, porém o armazenamento é menor

#### Estrutura de um frame



* **Constant pool:** Contém todas constantes necessárias para executar a classe, gerado pelo compilador. As constantes são metadados sobre a classe, variáveis com valores fixos em métodos, etc
* **Local variable array:** Guarda this, argumentos e as variáveis locais. O tamanho é definido pelo compilador
  + Em um método não static index 0 aponta para this, os index 1-n apontam para os argumentos
  + Em um método static os index 0-n apontam para os argumentos
  + De n até m são as variáveis locais, elas estão em ordem de declaração
* **Operand stack:** É uma stack FIFO que guarda operadores de operações matemáticas do método e os argumentos e retorno de métodos que serão chamados
  + Quando existe uma soma de x + y, x é adicionado e depois y, quando as variáveis são acessadas para a operação y é removido e depois x (a ordem original é conhecida), após a operação ser concluída o resultado é adicionado a stack
* **Frame data:** Guarda os metadados que serão necessários para executar o método, como referências para a constant pool, quando fazer o return e quando lançar exceptions

### Native method stack ou C stack

* Um **native method** é um método Java que é implementado em uma linguagem diferente de Java, como C ou C++.
  + Esses métodos são usados para acessar funcionalidades específicas do sistema ou APIs que não estão presentes na JVM
* A Native Method Stack funciona de maneira semelhante à JVM Stack, mas é usada para métodos nativos.
* De acordo com a especificação da JVM, uma implementação pode optar por não suportar métodos nativos, e, assim, não ter uma Native Method Stack.

### Program Counter (PC) registers ou call stack

* Um Program Counter (PC) é um registrador que guarda o endereço de memória e a thread da instrução sendo executada pela CPU.
  + Para um método **nativo**, o PC tem um valor indefinido, pois a execução ocorre fora da JVM.
* Quando a CPU finaliza a execução da instrução, o PC é incrementado para apontar para a próxima instrução.
* Cada thread possui um PC

## Como Objetos são Guardados em Memória

* **Heap:** Usado para objetos criados com new(), incluindo instâncias de classes e arrays.
  + Sempre que new() é utilizado ou um literal de string ("") é declarado, a JVM aloca memória no heap.
* **Stack:** Armazena referências a objetos no heap, valores de variáveis locais (primitivos), e contextos de execução de métodos

## Variáveis e tipos

* Atualmente os tipos são verificados pelo compilador antes de executar a aplicação, mas esse nem sempre foi o caso

### Primitivos

* Tipos primitivos guardam um valor
* Primitivos podem viver na stack como variáveis locais ou no heap como variáveis de instância (atributos)
  + Primitivos que vivem na stack são removidos da memória assim que a stack é removida

### Tipos Referência

* Tipos referência guardam referências (ponteiros) para um objeto, classe ou interface no heap
* Wrapper classes e String são tipos referência
* Tipos referência podem ser acessados de qualquer lugar, desde que seu endereço seja conhecido
* Referências podem viver na stack como variáveis locais ou no heap como variáveis de instância
* Os tipos referência podem ser divididos em class, array, interface e null
  + Uma referência para uma classe abstract apontará para a classe em si e não para um objeto que a implementa, o mesmo vale para interfaces
  + Um objeto é nada mais que uma representação em memória de uma classe acessada por uma referência

### Call by value

* Consiste em fazer a cópia de um tipo primitivo ou tipo referência quando passando um argumento ou retornando um valor de um método
* A cópia de um tipo primitivo é uma cópia do valor em si, de modo que o primitivo copiado é considerado um novo primitivo
* A cópia do tipo referência é a cópia da referência, de modo que a nova referência aponta para o mesmo objeto
  + As duas referências vão existir e é possível modificar o objeto a partir de ambas
  + Também pode ser chamado de shallow copy

### Escaping reference

* Ocorre quando a referência de um objeto B é obtida através de outro objeto A (composição) e isso permite editar o objeto B, mesmo que ele não deva ser alterado
* Ex: Um objeto Foo possui um campo private do tipo Bar (que é imutável, em tese) com um getter e sem setter,
  + Usando Foo.getBar() é possível obter uma referência para a instância de Bar e modificar essa instância, violando o encapsulamento de Foo e pode causar comportamentos inesperados
* **Immutabilty**: Escaping reference não ocorre com String e outros tipos imutáveis já que para alterar o valor de um desses objetos será necessário criar uma referência e fazer o objeto apontar para ela com Foo.setBar()
  + Wrappers, BigDecimal, BigInteger, etc são imutáveis
* **Deep copy:** Criar um objeto com uma nova referência, mas com os mesmos valores
  + Deep copy pode ser usado para resolver escaping reference já que as mudanças no objeto não afetarão a referência original, o custo é ter que alocar mais memória
  + No caso do exemplo getBar retornaria um new Bar() com os mesmos valores