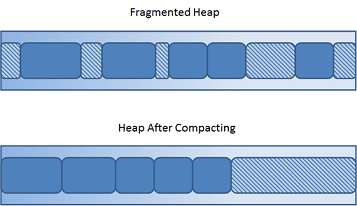
## Garbage Collector

* **Controle de Coleta:** A JVM solicita o garbage collection, mas o GC decide quando iniciá-lo. Métodos como Runtime.gc() e System.gc() podem solicitar à JVM que inicie a coleta, mas são apenas sugestões.
* **Impacto no Desempenho:** A coleta de lixo é um processo caro e dependendo da implementação pode pausar a execução do programa
* **Garbage Collector Tuning:** Ajusta e otimiza o comportamento do GC para minimizar pausas e melhorar a performance.
* **Fragmentação**: Problema de gerenciamento de memória que ocorre quando o espaço de memória disponível é dividido em blocos de espaços não contínuos, é divido em:
  + **Externa**: O heap tem muitos blocos pequenos e não contínuos de memória livre, ocorre após a alocação e liberação de vários blocos. Mesmo que exista espaço total disponível para alocar um objeto esse espaço pode estar disperso em vários blocos fazendo com que não haja memória suficiente
  + **Interna**: O espaço alocado para um objeto é maior do que o necessário, o espaço que foi alocado e não é usado é desperdiçado, causando ineficiência
  + O GC é responsável por lidar com fragmentação externa através de compactação (mover objetos, sem que fique espaço entre eles)



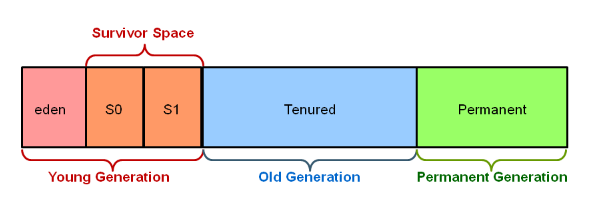
* **Live objects:** São objetos que não são elegíveis para remoção pelo GC (são referenciados em algum lugar da stack)
* **Mark e Sweep:** São as duas fases básicas do GC, mark é o processo de marcar objetos como live e sweep é o processo de remover objetos que não foram marcados (dead)

## Generational garbage collection

* Técnica que explora a curta vida da maioria dos objetos, dividindo o heap em áreas baseadas na idade dos objetos, priorizando a coleta de lixo nas áreas com objetos criados mais recentemente
* Possui várias implementações como **Concurrent Mark-Sweep (CMS)**, **Garbage First (G1)**, **ZGC**, e **Shenandoah GC, não necessariamente uma implementação vai seguir a risca**
  + **CMS, ZGC e Shenandoah usam algumas partes do conceito**
  + **G1 usa o conceito de maneira adaptada**

### Generational areas

* **Young Generation:** Onde a maioria dos objetos é alocada inicialmente
  + **Eden Space:** Armazena objetos recém-criados que ainda não passaram pelo GC.
  + **Survivor Space:** Armazena objetos que sobreviveram a coletas anteriores. É subdividido em S0 e S1
* **Old Generation/Tenured:** Guarda objetos que sobreviveram a múltiplas coletas na Young Generation. A coleta é menos frequente e mais demorada.
  + Objetos saem de um survivor space após n coletas e vão para cá.
* **Permanent Generation (antes do Java 8):** Armazena metadados da aplicação, como classes e métodos, possui tamanho fixo
* **Metaspace (a partir do Java 8):** Substitui a Permanent Generation, crescendo dinamicamente conforme a memória disponível e usa memória nativa em vez de virtual



### Tipos de Coleta

* **Class Unloading:** É o GC que ocorre no metaspace ele consiste em remover class loaders que não são referenciados (no heap) e metadados de classes que não estão sendo usadas (no metaspace). Essa coleta só ocorre quando o metaspace precisa expandir além de um limite ou quando ele está quase cheio

#### ****Major GC (Full GC):****

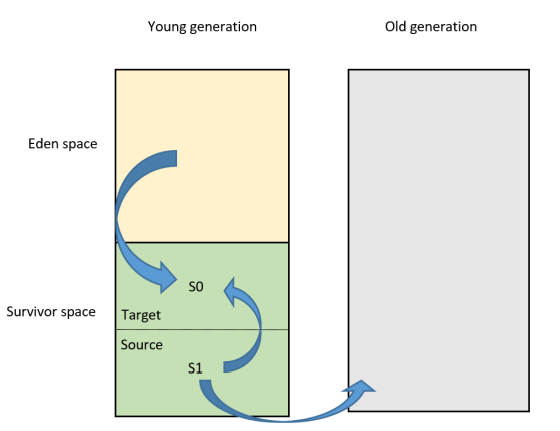
* Ocorre na Old Generation quando ela está full ou quase, sendo mais custoso e demorado. Visa limpar objetos de longa vida. Durante esse processo, as operações podem ser pausadas.

#### ****Minor GC (Young GC):****

* Ocorre quando o eden fica cheio, ou seja, quando a JVM não tem espaço para alocar a um objeto que foi criado
* **Idade**: A quantidade de vezes que um objeto sobreviveu ao GC
* **Ternuring threshold:** A idade mínima que um objeto deve ter para que ele seja movido para a old gen
* **Source survivor:** survivor space de onde objetos serão copiados. É a survivor zone com objetos
* **Target survivor:** survivor space para onde objetos serão copiados. É sempre a survivor zone vazia

##### Algorítimo

1. Live objects de eden são copiados para a target survivor e tem idade = 1
2. Live objects da source são examinados
   * Se um objeto for elegível ele será copiado para a old gen
   * Se não ele será copiado para a target e terá a idade incrementada
3. A memória da source survivor e do eden é limpa por completo (todos os objetos live estão salvos na target survivor)
4. No próximo ciclo a source estará vazia e será usada como target e target conterá objetos e será usada como source



* + Objetos live do eden são copiados para S0 e ficam com idade = 1
  + Objetos live da S1 são avaliados e copiados para old gen ou para a S0
  + Todos objetos de S1 e eden são removidos já que eles estão salvos em S0
* É possível usar um argumento na JVM que define que objetos com um tamanho superior a X devem ser mandados direto para a old gen
* Também é possível definir o tamanho das gerações usando argumentos da JVM,
  + No geral a young gen deve ter 25% a 33% do tamanho da old gen, garantindo que a old gen seja sempre mais larga, fazendo com que a maioria do GC ocorra na young gen, onde o processo é menos custoso

## Outros Conceitos

### GC Roots

* São os locais onde o processo de mark do GC começa, todos os objetos alcançáveis a partir de um GC root por uma referência direta ou indireta são live objects
  + Uma referência direta pode ser uma variável declarada dentro de um gc root
  + Uma referência indireta pode ser um objeto que está dentro de um objeto referenciado pelo GC root
* Um GC root normalmente é uma classe carregada no Class loader, parâmetros e variáveis de uma stack local, threads ativas ou objetos com native methods
  + O caso de native methods é especial já que não há como saber se algo é referenciado ou não por não ser código Java
* Um objeto não alcançável por um GC root não pode ser usado, já que sua referência de memória não é conhecida

### Marking

* Processo de GC que define quais objetos são live ou não

#### Tracing tracking

* Percorre todos os objetos a partir de um GC Root e altera um bit para 1 que define que o objeto é live, todo objeto que não foi marcado (bit = 0) será removido
* Todas implementações de GC usam uma variação de trace tracking
* Todos os objetos começam unmarked, ou seja, se o processo de marking perder um objeto ele será removido
* **Possível remoção de live object:** Se marking for feito em paralelo com a execução novos objetos podem ser criados em um GC root onde já foi feito marking, uma vez que objetos são criados com mark = 0, esse objeto será removido mesmo que seja um live object

##### Stop-the-world event

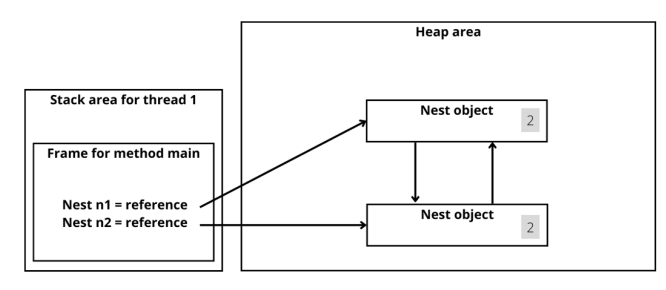
* Um stop-the-world event é um evento que causa a pausa de todas as threads. Dependendo do tempo de pausa o impacto no desempenho é grande
* Uma forma de evitar o problema de marking é por meio de stop-the-world events, a ideia é pausar a execução até que o marking seja feito, garantindo que nenhum objeto novo seja criado

#### Reference counting

* É uma alternativa a heap walking que evita a possível remoção de living objects sem causar stop-the-world event
* É usado um contador que diz quantas vezes um objeto é referenciado, esse contador é associado ao objeto de alguma forma.
  + Se o contador > 0 o objeto é considerado live
* Até o momento não existe nenhum GC que use reference counting

##### Islands of isolation

* É um problema de reference counting que ocorre quando objetos possuem uma referência mútua, de modo que um objeto Foo referencia Bar e um objeto Bar referencia Foo



* Apesar dos objetos serem referenciados uma vez pela stack, eles possuem um contador de 2, um para a referência do stack e outra para a referência mútua
* Quando esses objetos se tornarem inalcançáveis pela stack eles ainda serão considerados como live, pois cada um terá um contador de um, causando um memory leak eventualmente
* Esse problema possui algumas soluções, mas não entrarei em detalhes.

### Sweeping

* Remove objetos que não foram marcados por no processo de marking da memória
* Pode ser dividido em:
  + **Normal sweeping:** Remove objetos não marcados sem fazer nenhuma movimentação nos live objects, fazendo com que ocorra fragmentação
  + **Sweeping with compacting:** Remove objetos e depois faz a compactação da memória, evitando fragmentação. Uma vez que todos os blocos de memória precisam ser movidos é um processo custoso.
  + **Sweeping with copying**: São necessárias duas regiões de memória, uma região é usada para salvar todos blocos de memória que serão usados (live objects) e a outra tem todos seus blocos de memória removidos.
    - É o sweep citado anteriormente em minor GC
* Normal sweeping Só é recomendo quando existe muita memória livre e a prioridade é liberar memória rapidamente, em outros casos deve ser usado uma das alternativas
* O tipo de sweep depende da implementação de GC sendo usada