## Metaspace e Class Loading

* O metaspace guarda todos os metadados necessários para a JVM carregar uma classe:
  + Class files
  + Estrutura e métodos
  + Constantes
  + Anotações
  + Otimizações

### Class loading

* **Classpath:** É um conjunto de diretórios onde a JVM ou o compilador podem encontrar classes Java para serem executadas ou compiladas.
  + Maven consegue adicionar libs no projeto através de um class path
* Class loader é um objeto responsável por carregar uma classe em memória. Existem 4 tipos de class loader:
  + **Bootstrap class loader:** Responsável por carregar classes que vem com a JDK (incluindo os outros class loaders), é instanciado quando a JVM é inicializada.
  + **Extension class loader:** Carrega extensões usadas nas classes JDK (em $JAVA\_HOME/lib/ext). São extensões da JDK e não tem correlação com extensões da aplicação.
  + **System class loader:** Carrega as classes encontradas no classpath, herda de extension class loader
  + **Custom class loader:** É possível criar sua própria versão de class loader
* Quando uma classe do classpath é acessada pela primeira vez (ex: um objeto é criado), um system class loader é instanciado pela e JVM e é usado para encontrar o class file através do qualified name e depois salvar os metadados da classe no metaspace
  + Os metadados no metaspace são de propriedade do class loader
  + A instância do class loader é salva no heap
  + Uma vez que o class loader é instanciado ele será reusado
* **Dynamic class loading:** Cria instâncias de class loaders usando código Java, permitindo adicionar novas classes em tempo de execução, sem a necessidade de compilar o código novamente (método Class.forName(<“qualified name”>)
  + O uso mais conhecido é para fazer hot reloading
  + Dynamic class loaders e as classes que eles carregam são elegíveis para GC quando ambas não são alcançáveis
  + O class loader normal também é chamado de static class loader

### Memória no metaspace

* O metaspace pode crescer dinamicamente por ser armazenado em native memory
* Não existe GC no metaspace. Por usar de memória nativa e não virtual a memória do metaspace deve ser liberada por meio de interfaces da JVM.
* O GC é feito nos class loaders no heap e quando eles são removidos da memória no sweep os metadados de classe no metaspace são removidos
  + O class loader é considerado live quando objetos que foram criados através dele são live, ou seja, só ocorre sweep de um class loader quando todos objetos criados por ele são dead
* Um full GC é iniciado quando a memória disponível para o metaspace acaba ou quando um limite de tamanho (high-water mark) definido na JVM é atingido
  + A ideia é remover dead class loaders e depois liberar a memória no metaspace

## Configuração e monitoramento da JVM

* O modo que a JVM faz o gerenciamento de memória pode ser configurado de várias formas, de modo que algumas configurações vão aumentar o desempenho de algumas aplicações (JVM tuning)
* A regra principal sobre JVM tuning é que essa é a última opção para melhor do desempenho da aplicação. Se o código e o hardware podem ser melhorados isso deve ser considerado antes de JVM tuning
* JVM tuning consiste em monitorar a aplicação para entender o que precisa mudar, definir os novos parâmetros (VM args) e depois monitorar a execução com os novos parâmetros

### Métricas relevantes

* Existem várias métricas, essas são as mais relevantes
* **Well-functioning memory usage**: É a quantidade de memória que a aplicação precisa para rodar sem nenhum problema enquanto ela está funcionando normalmente.
  + É a quantidade de memória mínima que a aplicação usa em um estado estável mais uma quantidade extra, de modo que a memória não esteja sempre cheia ou quase cheia.
  + Alocar toda memória do SO não é uma boa ideia já que essa memória é necessária para o SO executar tarefas básicas/outros processos.
* **Normal latency:** Latência é o tempo que uma aplicação demora para responder (ex: tempo para processar request HTTP)
  + Dependendo da aplicação é difícil medir pois problemas de rede podem interferir (ex: a rede demora para enviar uma request para outra aplicação)
* **Level of throughput:** throughput é a quantidade de trabalho que pode ser feita em um certo período de tempo, quanto maior o level, melhor. Altos leveis usam mais memória e podem afetar a latência

### Profilling e sampling (monitoramento)

* **Profiling:** Monitora a execução e notifica um observador quando um evento ocorre (ex: memória chegou a x%), fornece mais informações, mas afeta o desempenho
* **Sampling:** Faz thread dump (snapshot do estado de todas threads de um processo java) para que ele possa ser analisado posteriormente
* Ambos consistem em coletar dados da aplicação enquanto ela está rodando para análise
  + O monitoramento da JVM pode causar impactos no desempenho, sendo recomendado não fazer em produção
  + jstat e jhsdb são comandos que vem com a JDK e permitem fazer o monitoramento, VisualVM é uma ferramenta para o mesmo propósito. Intellij também suporta

### Configuração do heap size

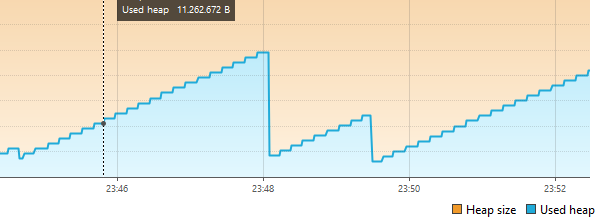
* Quanto menor o heap size mais ciclos de GC ocorrerão, e quanto maior o heap size maior o tempo de cada ciclo de GC
  + Uma boa medida é gastar menos de 5% do tempo de execução fazendo GC
* Normalmente o heap size é configurado para ocupar até 50% da memória disponível, garantindo que os processos do SO tenham memória
* Xms1024m define o tamanho inicial do heap
* Xmx1024m define o tamanho máximo do heap
* Xss1024m define o thread stack size
* Heap dump: Mesma coisa de thread dump, mas no heap, mostra os objetos em memória é feito com -XX:+HeapDumpOnOutOfMemoryError

### Configuração do metaspace

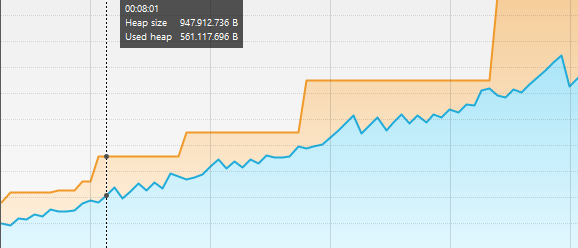
* -XX:MaxMetaspaceSize=2048m para definir o tamanho máximo esperado
  + Se o metaspace chegar ao tamanho ele vai tentar fazer GC para se manter no tamanho ideal e se não conseguir vai expandir
* -XX:MetaspaceSize=1024m limite para iniciar o GC
* -XX:MinMetaspaceFreeRatio=50 e -XX:MaxMetaspaceFreeRatio=50: A JVM vai tentar manter a taxa de espaço livre entre min% e max%, fazendo resize do metaspace
  + Fazer o resize da memória para o metaspace é custoso para o CPU, usar uma min/max alto é uma boa ideia quando se pretende carregar muitas classes ao mesmo tempo

### Configuração do GC

* Um heap com uma coleta eficiente vai subir e descer, seu gráfico vai ser parecido com os dentes uma serra.
* O consumo de memória vai de um ponto inicial e sobe até iniciar um ciclo de coleta e então cai, próximo do inicial



* Se o gráfico de consumo começa em um ponto n, sobe e volta para um ponto um pouco mais elevado que n, fazendo com que o consumo cresça com o tempo, isso pode ser um indicador de memory leak



* Outro indicador de memory leak são muitos ciclos de GC, mas sem grandes efeitos
* Um GC não vai consertar um memory leak, no melhor dos casos vai fazer com que o OutOfMemoryError demore mais para acontecer
* Se o consumo de memória for elevado e não houver memory leak faz sentido considerar outra implementação de GC

## Evitando Memory Leaks

* Ocorre quando um objeto que não é necessário para a execução não é liberado
* Conforme a memória fica cheia por causa de um memory leak a aplicação começara a ficar lenta até que um OutOfMemoryError cause a interrupção da JVM
* O tempo até que OutOfMemoryError ocorra depende de quantas vezes o código que causa o memory leak é executado.
* O maior sintoma de um memory leak é a lentidão
* A regra de ouro é: Objetos que não são necessários não devem ter nenhuma conexão direta ou indireta com a stack

## Evitando erros comuns de memória no código

* **Manter referências não usadas:** Um erro comum é manter referências para objetos que não estão sendo usados, o jeito mais simples de resolver isso é definindo objetos para null após seu uso ou atribuindo um novo valor a eles
  + Isso nem sempre é uma abordagem necessária nos softwares novos
* **Deixar recursos abertos:** Não fechar um recurso Closeable (arquivo, banco de dados, streams, etc) também causa problemas (ex: stream ficar cheia). Deve-se usar try-with-resources com esse tipo de recurso ou fechar manualmente
* **Criar strings desnecessárias:** Cada string criada ocupa memória, quando se concatena n strings com +, são criadas n strings em memória. Quando se precisa concatenar muitas strings é recomendável usar StringBuilder
* **Usando wrappers desnecessariamente:** Wrappers requerem mais memória que primitivos. Em alguns casos é inevitável usar wrappers, mas sempre que possível devem ser usados primitivos
* **Static collections:** Usar static collections não é recomendável pois os objetos ficaram live através da classe indefinidamente. Se for necessário usar static collections elas devem ter um tamanho limitado e/ou os objetos devem ser gerenciados manualmente.