# Garbage Collector in Java

*how your VM does the "chores" instead of you*

# Questions to answer

- **What** is a garbage collector?
- **What approaches** a garbage collector uses?
- **How** is the JVM's garbage collector working?
- **How** can we improve GC's performance?

# **What** is a garbage collector?

- **Memory management** handles where and how objects are represented in memory
- **Problem:** objects not used anymore occupy space
- **Solution:** remove them !!!
- **Ways:** explicit (*C++ style*) vs automatic (*Java style)*

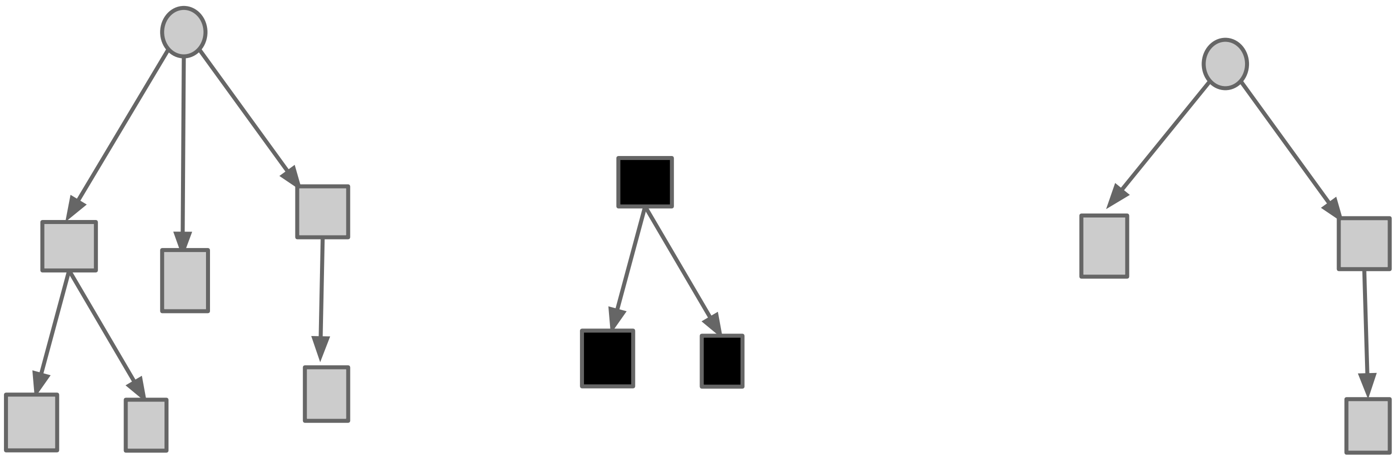
# **What** is a garbage collector?

- A **garbage collector** is an automatic memory management system
- Represent a set of **data structures** and **algorithms** hidden from the developer
- Main task is to find the objects that are no longer used and remove them

# Terminology

- *Root* - local or global variable
- *Live object* - object referenced either by a root or another live object
- *Garbage object* - object created by the program which is not live
- *Reachability* - an object is reachable from another one if there is a chain of references that links them

# Graph analogy



Root

Live Object

Garbage

# Design choices

- Serial vs Parallel
- Concurrent vs Stop-the-world
- Compacting vs Non-compacting vs Copying

# Performance metrics

- Throughput
- Overhead
- Pause time
- Frequency of collection
- Footprint
- Promptness

# Basic algorithms: Mark-and-Sweep

- A 2-phase algorithm
- **Mark phase** - a DFS search from roots to find all the live objects and mark them
- **Sweep phase** - visiting all objects and deleting those unmarked
- **Compacting option** - moving live objects left most
- **Performance**: linear in terms of no of objects
- **Problems**: fragmentation*

# Basic algorithms: Copying

- A 1-phase algorithm
- There are 2 equal memory allocation spaces: **old space** and **new space**
- During the **mark phase** live objects are copied from **old space** to **new space**
- At the end algorithm the roles of the physical spaces are swapped
- Fast but cuts the available space by half !!!

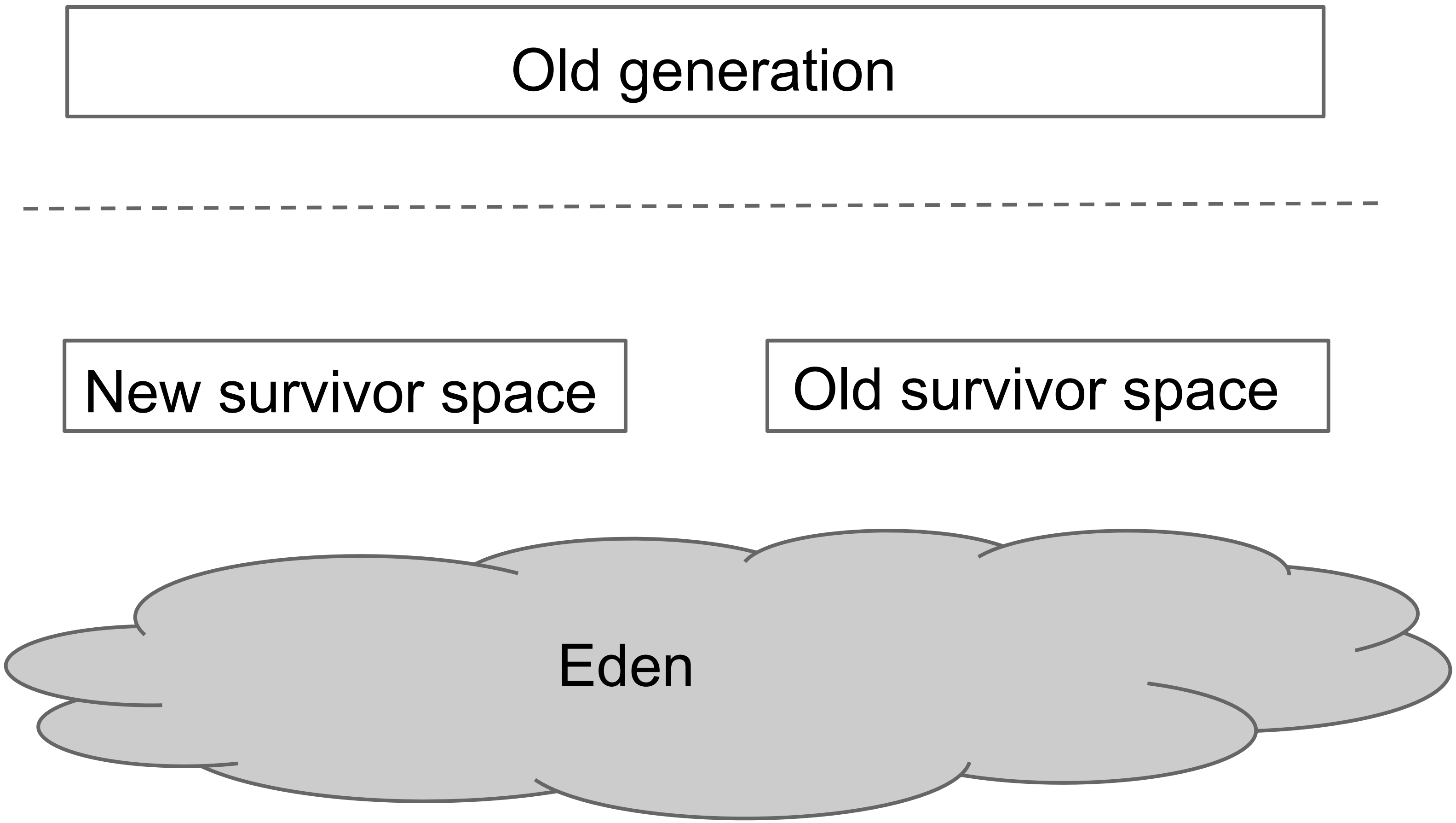
# Basic algorithms: Generational

- **Idea**: the longer one object lives the more probable is the object would live longer
- At least 2 **generations** of objects can be observed
- **Young generation** - objects that have a short life (usually small)
- **Old generation** - objects that have a long life (usually larger)
- Algorithms tuned for each generation can be used

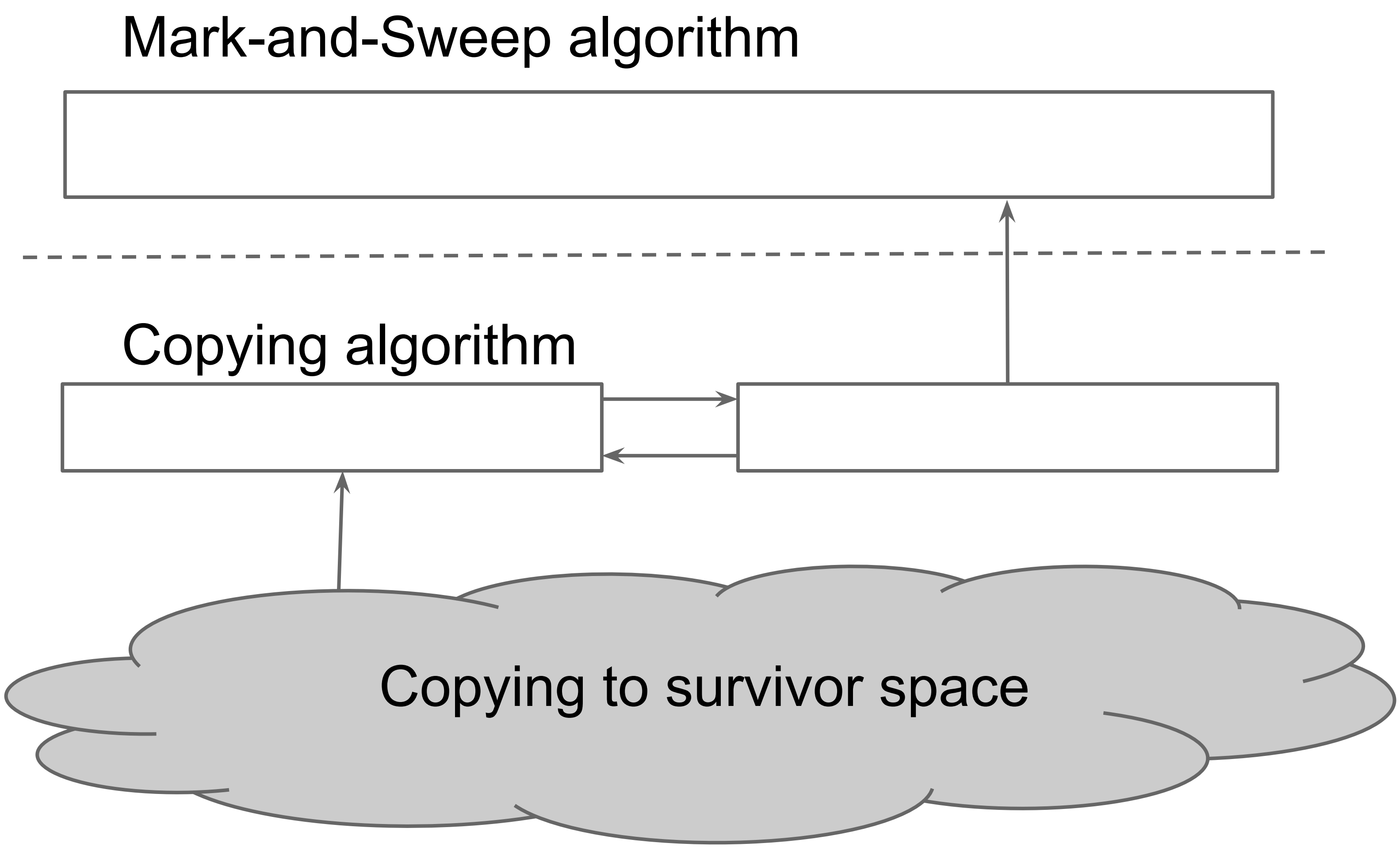
# HotSpot VM Generational Model

- Uses 3 generations: **young**, **old** and **permanent**
- **Young generation** contains *Eden* and *2 survivor spaces*
- Almost all the objects are allocated in *Eden* and then promote to *survivor spaces*
- **Old generation** contains objects that survived several collections
- **Permanent generation** info about classes and methods

# HotSpot VM Generational Model



# HotSpot VM Generational Model



# HotSpot VM Generational Model

- The objects are allocated in *Eden*
- When the *Eden* is collected the live objects are copied in the current *new survivor space* and the *Eden* is erased
- In the survivor space the copying algorithm is used
- After an object survived several collections in the survivor space is moved in old generation space
- In the old generation space the Mark-and-Sweep algorithm is used

# HotSpot VM Algorithms

- **Serial Collector**
  -XX:+UseSerialGC
- **Parallel Collector**
  -XX:+UseParallelGC
- **Parallel Compacting Collector**
  -XX:+UseParallelOldGC
- **Concurrent Mark-and-Sweep Collector**
  -XX:+UseConcMarkSweepGC

# HotSpot VM: Serial Collector

- The collections for both **new** and **old generations** are done serially with the execution of program
- **Stop-the-world** happens when the collector is running
- Useful for most applications run on a client-style machine (1 CPU) without constraints on pause time
- Enabled by default on client machines

# HotSpot VM: Parallel Collector

- Collection of young generation is done using **multiple threads** in parallel
- Collection of old generation is done serial
- Still a **Stop-the-world** algorithm but the **pause time** is decreased
- Decreases **overhead**, increases **throughput**
- Useful on multi-core machines for applications with medium to large data sets

# HotSpot VM: Parallel Compacting

- Same as parallel collector, just uses multiple threads for old generation collection too
- The **Mark-and-Sweep algorithm** is used with 3 phases
- **Mark phase** - parallelized
- **Summary phase** - serial
- **Sweep phase** - parallelized
- Better parameters for **throughput, overhead** and **pause time** than parallel collector

# HotSpot VM: Concurrent Collector

- Use the same algorithm for **young generation** as parallel collector
- Collection for **old generation** is done concurrently with the program execution
- The concurrent algorithm has 4 phases
- **Initial mark** - stop-the-world, serial
- **Concurrent marking**
- **Remark** - stop-the-world, parallel
- **Concurrent sweep**

# HotSpot VM: Concurrent Collector

- Is a **non-compacting algorithm** (leads to fragmentation - bad!!! )
- Overhead when allocating objects in old generation
- The collections starts before the heap is full
- **Floating garbage** can survive between collections
- Significantly decreases **pause time**

# HotSpot VM Selecting collector

- Use **serial algorithm** when:
  - app has small data set
  - 1 CPU available and no constraint on pause time

- Use **parallel algorithm** when:
  - performance is most important and pause time constraint is not very "heavy"

# HotSpot VM Selecting collector

- Use **parallel compacting algorithm** when:
  - the same conditions as for parallel algorithm where met and the machine has more than 2 CPUs

- Use **concurrent mark-and-sweep** when:
  - response time more important than throughput and pause times have to be <1s

# HotSpot VM Ergonomics

- Parameter tuning of algorithms is done using **automatic selection** and **behavioural tuning**
- **Automatic selection** selects the type of algorithm and parameters best for the hardware configuration
- A machine is considered **server** if it has 2 or more CPUs and 2 GB RAM or more

# HotSpot VM Ergonomics

- **Behavioural tuning** refers to setting a goal for one parameter and let the runtime tune its parameters to achieve it
- **Maximum Pause Time Goal**

  -XX:MaxGCPauseMillis = n
- **Throughput Goal**

  -XX:GCTimeRatio = n

  ratio of garbage time = 1/ (1+n)

# HotSpot VM Extra-tuning

- The best approach for extra-tuning is the **do-nothing approach !!!**
- Change the default collector or the behavioural parameters (if appropriate)
- Use debugging tools available in the JDK
- May want to modify sizes used in the algorithms: *-Xmx,-Xms, NewRatio, MinHeapFreeRatio, MaxHeapFreeRatio, SurvivorRatio*

# Code considerations

- The Java API has 2 functions: System.gc(), Runtime.gc()
- These functions **do not** force execution of GC, but merely gives a hint to the VM
- Methods to leverage the GC: **object pooling** and **use of weak references**

# Code consideration: Object Pooling

- Design pattern consisting of creating objects beforehand and keeping them alive during the execution of the program
- A **pool** is a set of objects ready to use without needing allocation
- Useful just when: initialization cost is high, rate of instantiation is high and instances in use at a time is low
- **Controversial** as modern GC are powerful

# Code considerations: Weak references

- Different types of references: **strong, soft, weak** and **phantom**
- **Weak references** do not count when considering reachability
- **Soft references** same as weak references but are not collected if can be saved in memory
- Weak and soft references are useful in situations like caching or dynamic relations between instances

# Sources


- UC Berkeley CS 61B: Data Structures - Fall 2006

  http://www.youtube.com/watch?v=rp8PvFvSO_c

  http://www.youtube.com/watch?v=zksIj9O8_jc
- Sun documentation

  http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf

  http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html
- weblogs.java.net

  http://weblogs.java.net/blog/2006/05/04/understanding-weak-references

# Questions?