Heaps and Garbage

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Final Exam

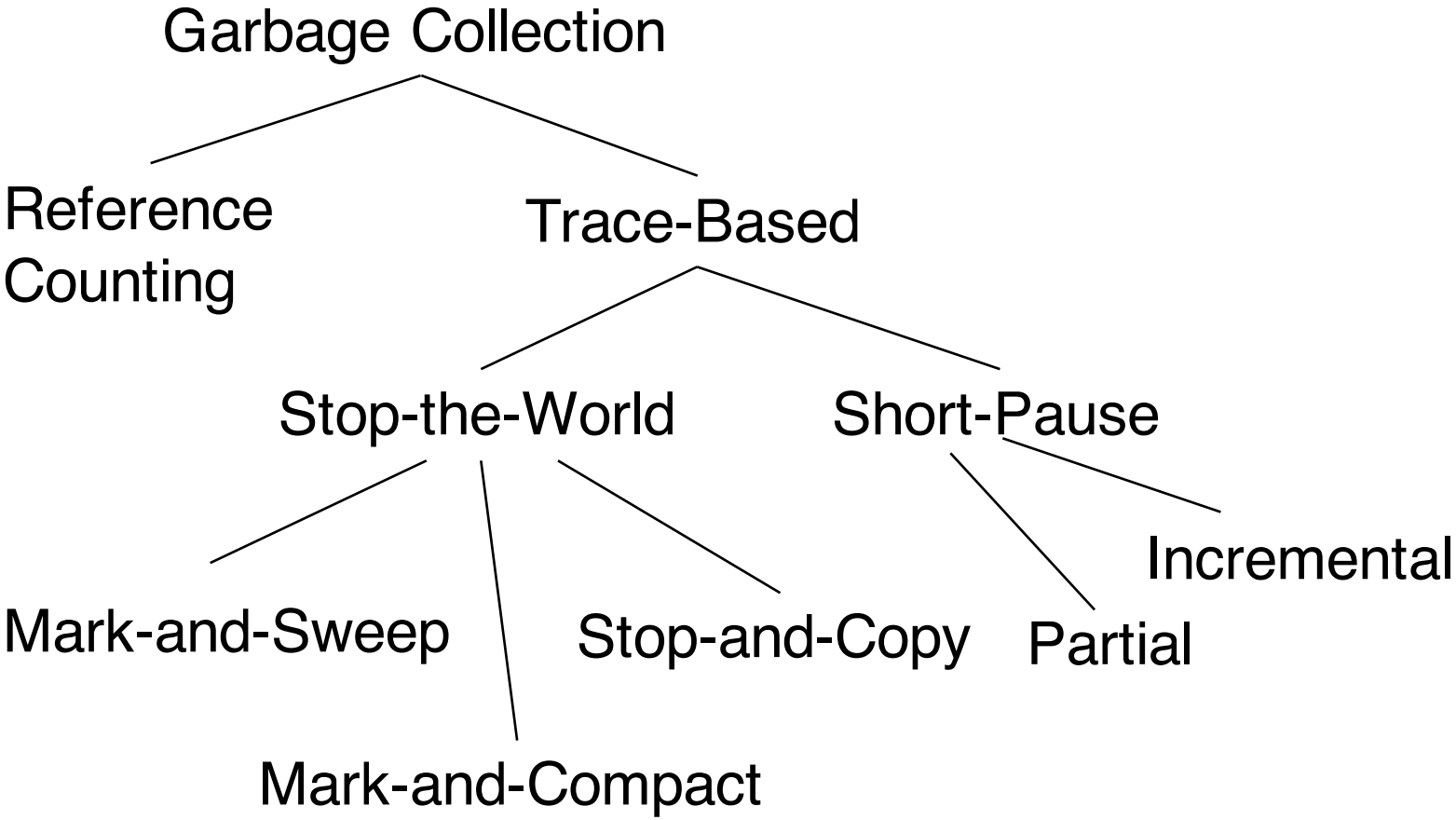
Cumulative (35% of final grade)

Dec. 14 (Wednesday) 6:50PM-8:40PM

119 Osmond Lab

Last assignment to be released soon (due on the last day of class)

Taxonomy


Garbage Collection
Reference Counting
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Stop-and-Copy
Partial
Incremental

Partial Collector

Add stable set to **root set**

Do GC as before (use a stop-the-world algorithm)

Note: garbage might survive the collection

Stop-the-World Collectors

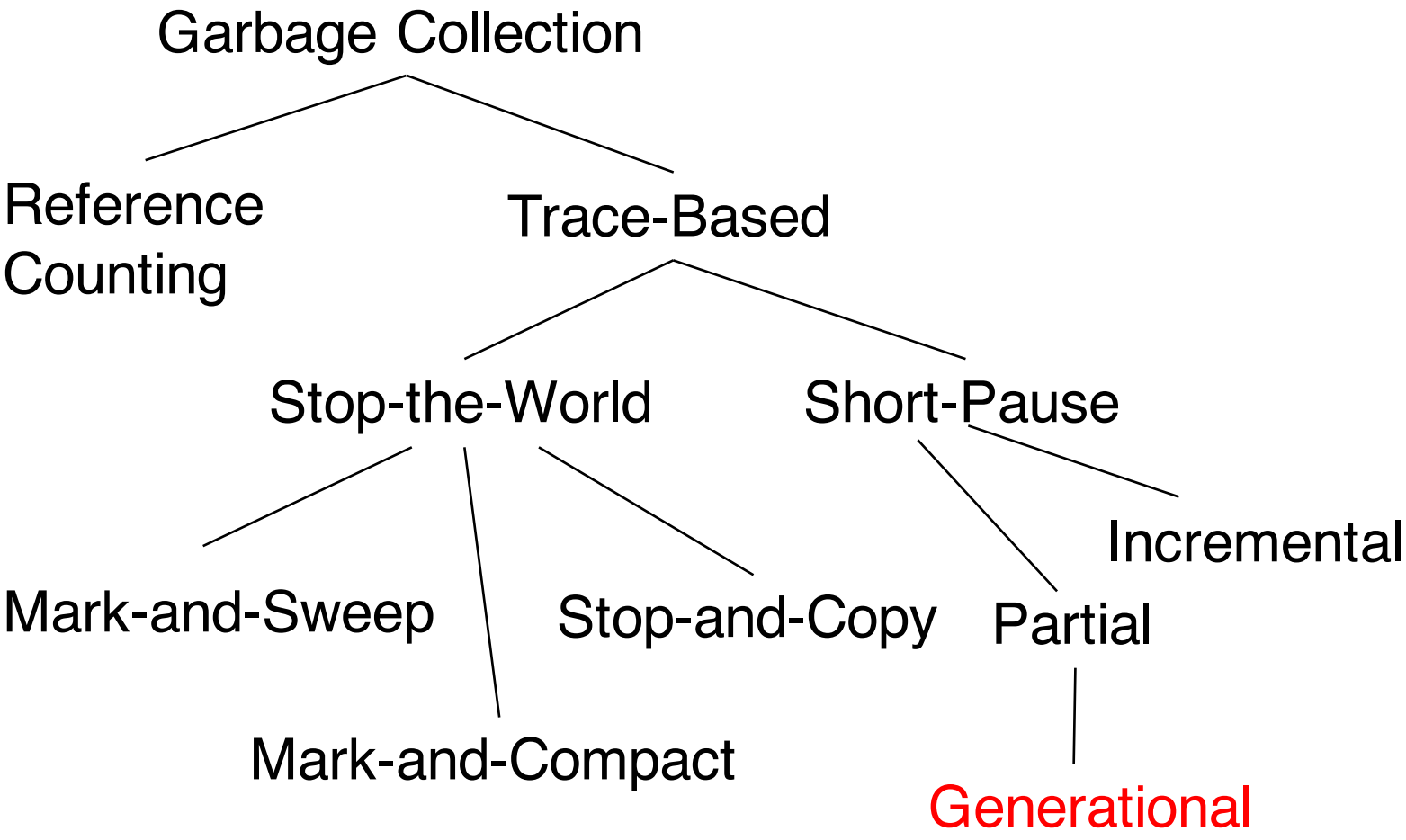
A: number of alive objects; N: all objects on heap

H: total heap size

	Mark-And-Sweep	Mark-And-Compact	Stop-And-Copy
Complexity	O(N)	O(N)	O(A)
Fragmentation?	Yes	No	No
Memory move	No	Yes	Yes
Effective heap size	H	H	H/2

A hybrid algorithm that gets the best of these collectors?

Taxonomy


Garbage Collection
Reference Counting
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Stop-and-Copy
Partial
Incremental
Generational

The Object Life-Cycle

“Most objects die young.”

- But those that survive one GC are likely to survive many

Idea: tailor GC to spend more time on regions of the heap where objects have just been created

A better ratio of reclaimed space per unit time

Generational

Divide heap into generations g_1, g_2 …

- g_i holds older objects than g_{i-1}

Create new objects in g_1, until it fills up

GC g_1 only; move reachable objects to g_2 after several collections (typically one collection)

Generational

When g_2 fills, garbage collect g_1 and g_2, and put the reachable objects in g_3

In general: When g_i fills, collect g_1, g_2, …, g_i, and put the reachable objects in g_{i+1}

What GC algorithm is better for young generation?
How about old generation?

Algorithm for Young Generation

A: number of alive objects; N: all objects on heap

$A \ll N$

	Mark-And-Sweep	Mark-And-Compact	Stop-And-Copy
Complexity	O(N)	O(N)	O(A)
Fragmentation?	Yes	No	No
Memory move	No	Yes	Yes
Effective heap size	H	H	H/2

Stop-and-copy is the most efficient

Algorithm for Old Generation

A: number of alive objects; N: all objects on heap

A is close to N

	Mark-And-Sweep	Mark-And-Compact	Stop-And-Copy
Complexity	O(N)	O(N)	O(A)
Fragmentation?	Yes	No	No
Memory move	No	Yes	Yes
Effective heap size	H	H	H/2

Mark-And-Compact removes fragmentation

Algorithm for Old Generation

A: number of alive objects; N: all objects on heap

What if most objects have the same size?

	Mark-And-Sweep	Mark-And-Compact	Stop-And-Copy
Complexity	O(N)	O(N)	O(A)
Fragmentation?	Yes	No	No
Memory move	No	Yes	Yes
Effective heap size	H	H	H/2

Mark-And-Sweep saves the cost of moving objects

Generational

Pros:

Divide heap according to lifetimes of data

Great for data with mixed lifetimes

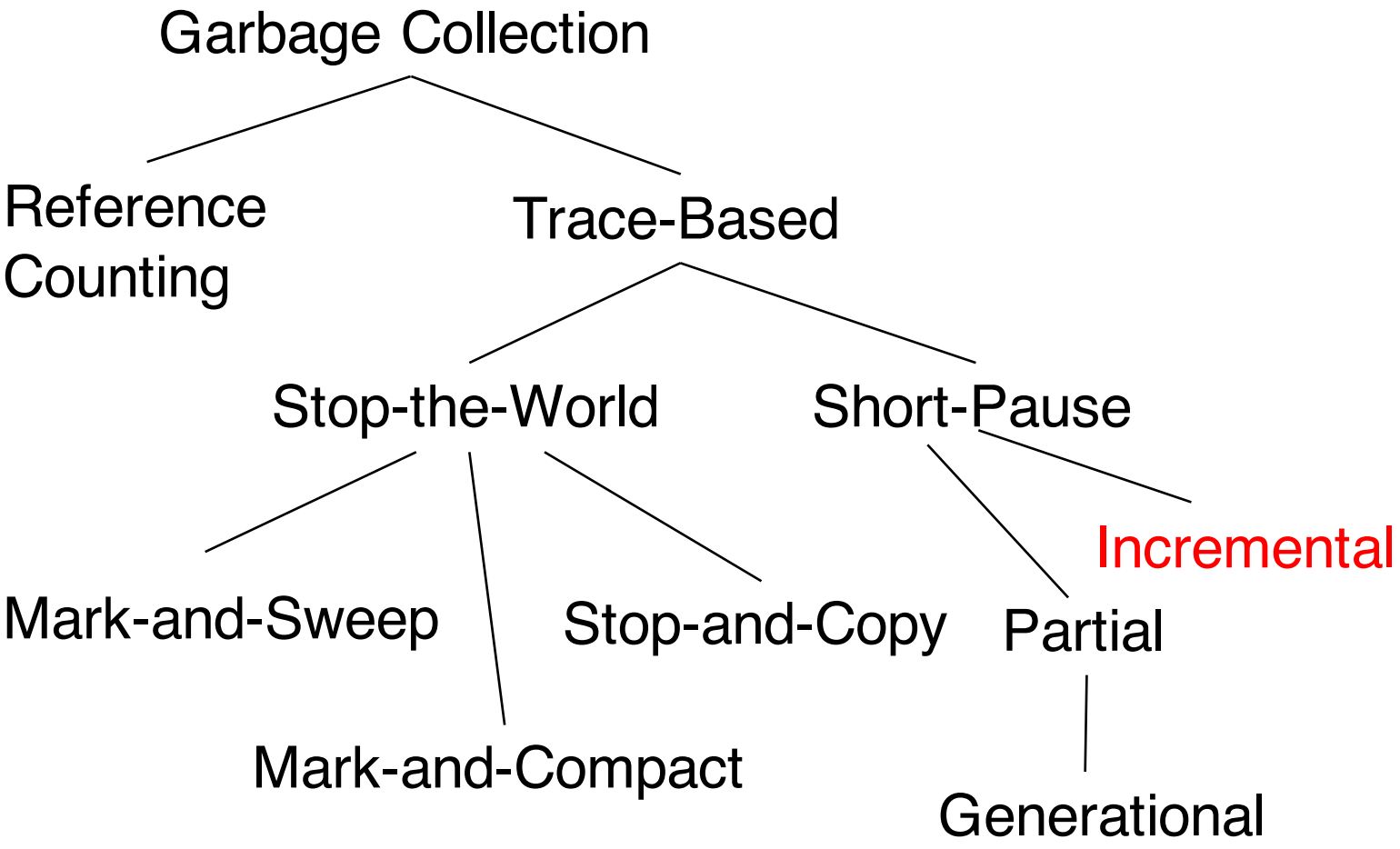
Cons:

Garbage might survive a collection

Small heap size for each generation

More frequent collection

Taxonomy


Garbage Collection
Reference Counting
Trace-Based
Stop-the-World
Short-Pause
Incremental
Mark-and-Sweep
Stop-and-Copy
Partial
Mark-and-Compact
Generational

Problem with Incremental GC

Run garbage collection in parallel with program

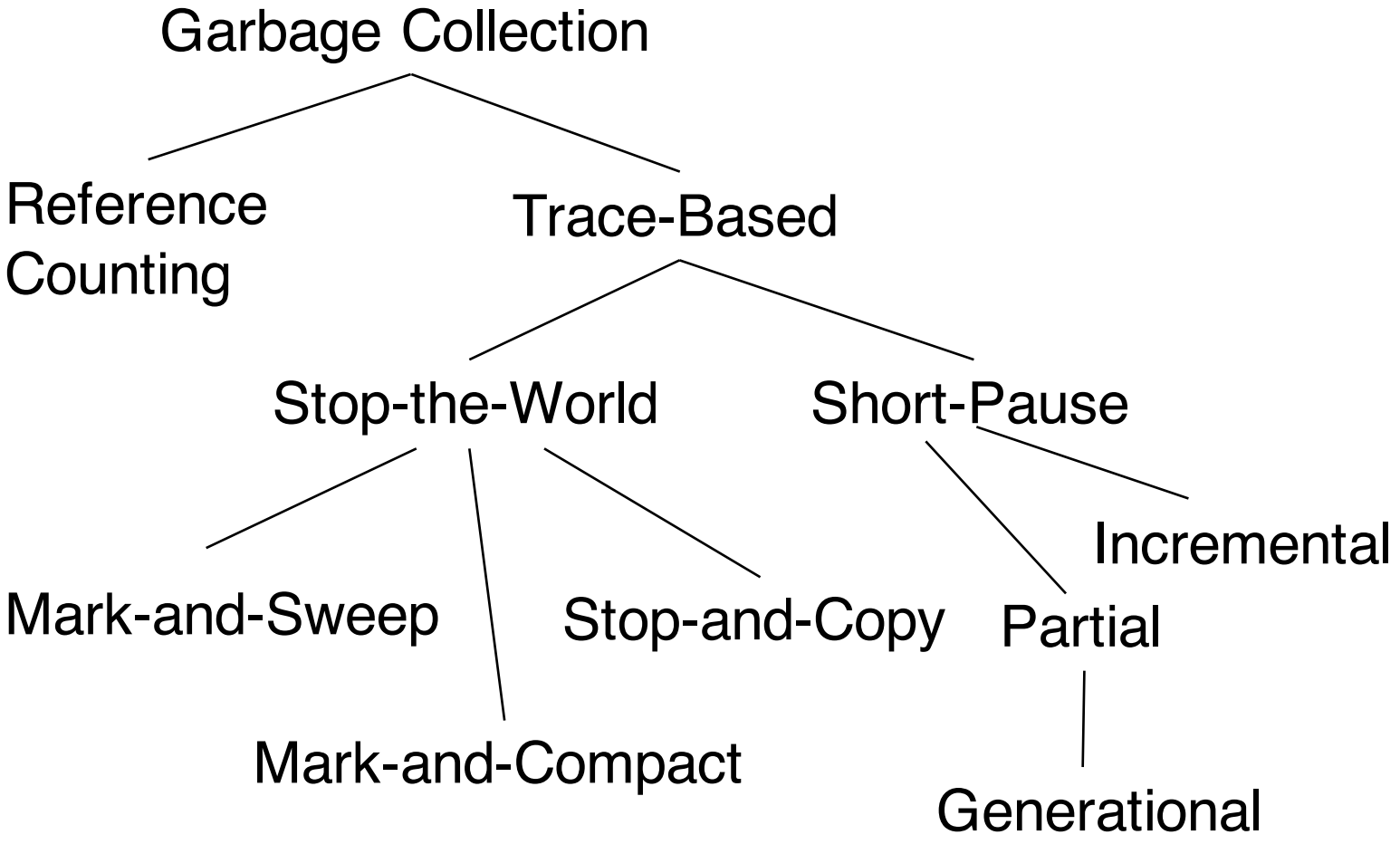
What's tricky?

If reference in a visited object is changed to point to an unreachable object, the latter may be collected

A trick (Write Barrier)

Protect all pages containing visited objects. A hardware interrupt will invoke collector to fix links

Taxonomy


Garbage Collection
Reference Counting
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Stop-and-Copy
Partial
Incremental
Generational

Reference Counting vs. Trace-Based

Reference counting collects garbage on-the-fly

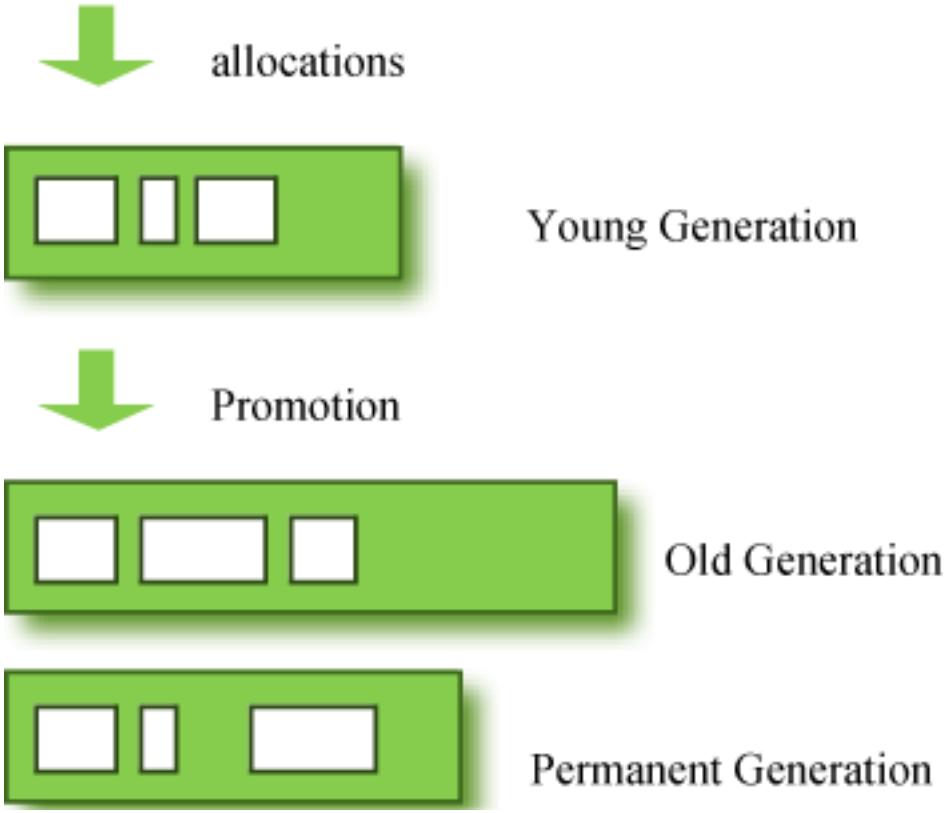
- Counter updated for each reference/dereference
- Collect object whenever its counter reaches 0
- Preferred for cycle-free tasks

Trace-Based is more general

- Incremental/Partial collection alleviates the stop-the-world issue

Case Study: Java SE

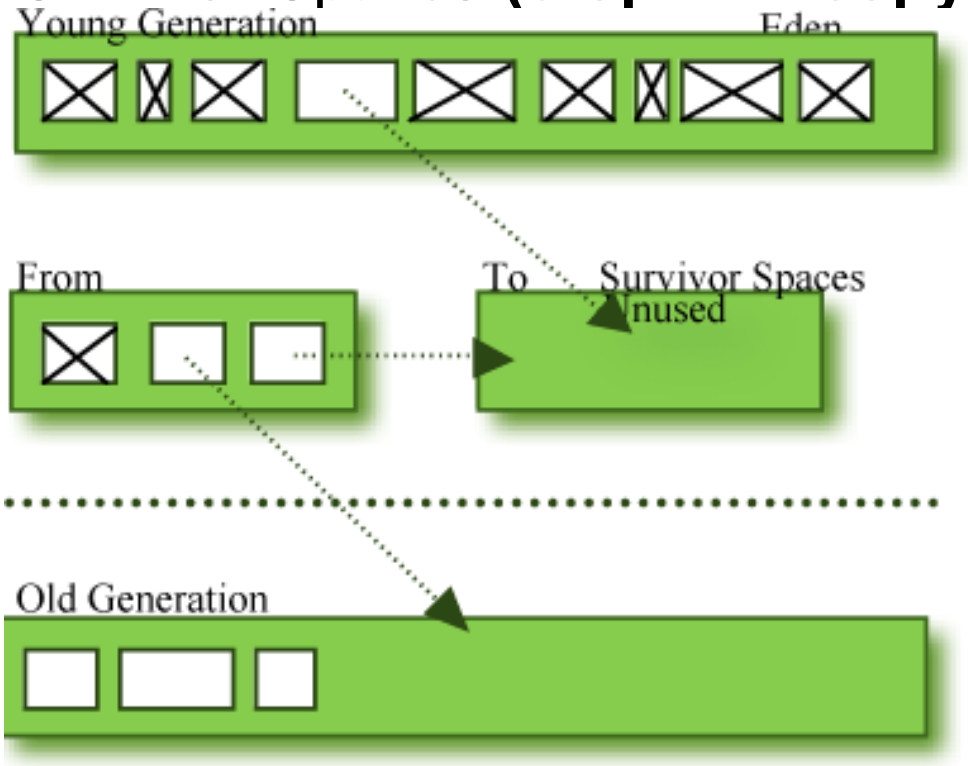
Generational Collectors



Case Study: Java SE

Generational Collectors

- Young Generation, 3 partitions:
  - Eden, 2 Survivor Spaces **(stop-and-copy)**



Case Study: Java SE

Generational Collectors

- Young Generation, 3 partitions:
  - Eden, 2 Survivor Spaces **(stop-and-copy)**
  - Minor collections (high mortality rate)
- Old Generation
  - surviving objects promoted from Young Generation
  - Major collections (infrequent)
- Permanent Generation
  - objects needed by JVM, Strings

Three Collectors: ***Serial, Throughput, Concurrent***

Three Implementation Approaches

Stop-the-World

- Used in ***serial*** and ***throughput***

Incremental

- ***Concurrent*** does this for minor & major collections

Partial

- Generational: 3 Generations

Serial Collector

```
-XX:+UseSerialGC
```

Uses a single thread to perform all GC

Relatively efficient since there is no communication overhead between threads

Best-suited to single processor machines

Throughput (Parallel) Collector

```
-XX:+UseParallelGC
```

Uses multiple threads for minor collection

Significantly reduce garbage collection overhead

Best-suited to multiple processor machines, application with a large number of short-lived objects, and no pause time constraint

Concurrent Collector

```
-XX:+UseConcMarkSweepGC
```

Concurrent GC with program (incremental)

Keep garbage collection pauses short

Best-suited to multiple processor machines, application with a large number of long-lived objects, and a pause time constraint