Comp-4400: Garbage Collection

November 25, 2019


2455
REPUBLIC
SERVICES
CORVALLIS, OR
USDOT 0167398
THRASH 'N' TRASH PRODUCTIONS
"Wasting Away Since 2005"

Outline

Motivating example

What will happen if we run the following program?

```
public static void main(String[] args) throws Exception{
        int m=100000;
        int n=10000;
        Double [][] A=new Double [m][n];
}
```

We can check the Heap size of your computer using the following command:

```
ianguos-MBP:code jianguolu$ java -XX:+PrintFlagsFinal -version | grep -i 'HeapSize'
    uintx ErgoHeapSizeLimit                         = 0
{product}
    uintx HeapSizePerGCThread                       = 87,241,520
{product}
    uintx InitialHeapSize                          := 268,435,456
{product}
    uintx LargePageHeapSizeThreshold                = 134,217,728
{product}
    uintx MaxHeapSize                              := 4,294,967,296
{product}
java version "1.8.0_71"
Java(TM) SE Runtime Environment (build 1.8.0_71-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.71-b15, mixed mode)
```

The Heapsize is 4GB, big enough for this program. What is really happening behind?

Motivating example

What will happen if we run the following program?

```
public static void main(String[] args) throws Exception{
        int m=100000;
        int n=10000;
        Double [][] A=new Double [m][n];
}
```

We can check the Heap size of your computer using the following command:

```
ianguos-MBP:code jianguolu$ java -XX:+PrintFlagsFinal -version | grep -i 'HeapSize'
    uintx ErgoHeapSizeLimit                         = 0
{product}
    uintx HeapSizePerGCThread                       = 87,241,520
{product}
    uintx InitialHeapSize                          := 268,435,456
{product}
    uintx LargePageHeapSizeThreshold                = 134,217,728
{product}
    uintx MaxHeapSize                              := 4,294,967,296
{product}
java version "1.8.0_71"
Java(TM) SE Runtime Environment (build 1.8.0_71-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.71-b15, mixed mode)
```

The Heapsize is 4GB, big enough for this program. What is really happening behind?

Motivating example

What will happen if we run the following program?

```
public static void main(String[] args) throws Exception{
        int m=100000;
        int n=10000;
        Double [][] A=new Double [m][n];
}
```

We can check the Heap size of your computer using the following command:

```
ianguos-MBP:code jianguolu$ java -XX:+PrintFlagsFinal -version | grep -i 'HeapSize'
    uintx ErgoHeapSizeLimit                         = 0
{product}
    uintx HeapSizePerGCThread                       = 87,241,520
{product}
    uintx InitialHeapSize                          := 268,435,456
{product}
    uintx LargePageHeapSizeThreshold                = 134,217,728
{product}
    uintx MaxHeapSize                              := 4,294,967,296
{product}
java version "1.8.0_71"
Java(TM) SE Runtime Environment (build 1.8.0_71-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.71-b15, mixed mode)
```

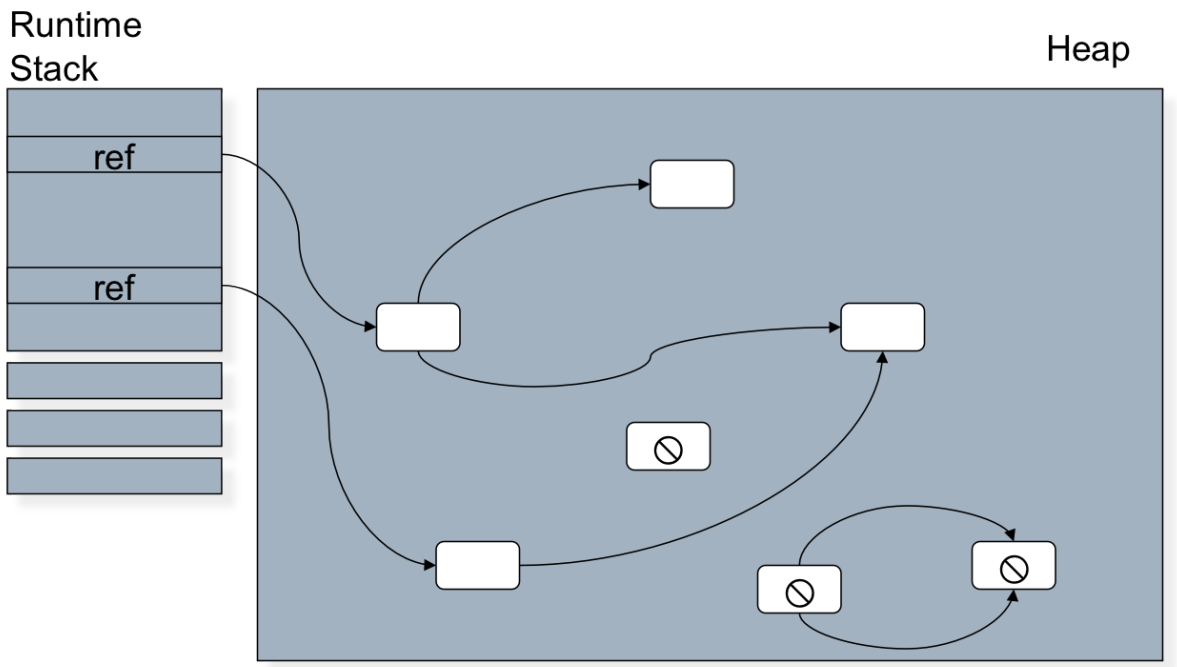
The Heapsize is 4GB, big enough for this program. What is really happening behind?

What is garbage collection

- Memory Management technique.
- Process of freeing objects no longer referenced by the program.

Fundamental Garbage Collection Property

“When an object becomes garbage, it stays garbage”



Dynamic Memory Allocation

- In Java, “new" allocates an object for a given class.

```
1   Account account = new SavingsAccount("1234567");
```

- But there is no instruction for manually deleting the object.
- Objects reclaimed by a garbage collector when the program “does not need it" anymore.

```
1   account=null;
```

- Cases that objects are no longer used:
  - All references to that object explicitly set to null e.g. object = null
  - The object is created inside a block and reference goes out scope once control exit that block.
  - Parent object set to null if an object holds the reference to another object and when you set container object’s reference null, child or contained object automatically becomes eligible for garbage collection.
- “The Garbage Collection Handbook: The Art of Automatic Memory Management" by Richard Jones, Anthony Hosking, and Eliot Moss.

Manual vs automated memory management

Manual allocation and de-allocation

```
1  ptr=malloc(256 bytes);
2  /* use ptr */
3  free(Ptr);
```

- manual memory management: let programmer decide when objects are deleted.
- automatic memory management: let a garbage collector delete objects.

Problems of manual management

- Manual memory management creates severe debugging problems
  - Memory leaks,
  - Dangling pointers.
- In large projects where objects are shared between various components it i sometimes difficult to tell when an object is not needed anymore.
- Considered the big debugging problem of the 80's

Solution: Automatic Memory Reclamation

users allocate space for an object

```
1  Account account = new SavingsAccount("1234567");
2
3  for (File f : files) {
4      String s = f.getName();
5  }
```

system collects the space when not used

- When the system “knows" the object will not be used anymore, it reclaims its space.
- Telling whether an object will be used after a given line of code is undecidable.
- Therefore, a conservative approximation is used.
- An object is reclaimed when the program has “no way of accessing it".
- Formally, when it is unreachable by a path of pointers from the “root" pointers, to which the program has direct access.
- Local variables, pointers on stack, global (class) pointers, JNI pointers, etc.
- It is also possible to use code analysis to be more accurate sometimes.

What’s good about automatic “garbage collection"?

Software engineering

- Relieves users of the book-keeping burden.
- Stronger reliability, fewer bugs, faster debugging.
- Code understandable and reliable. (Less interaction between modules.)

Security (Java)

Program never gets a pointer to ”play with”.

GC and languages

Sometimes it's built in:

- LISP, Java, C #.
- The user cannot free an object.

Sometimes it's an added feature:

- C, C++.
- User can choose to free objects or not. The collector frees all objects not freed by the user.

Most modern languages are supported by garbage collection.

what is bad about automated GC

It has a cost:

- Old Lisp systems 40%.
- Today's Java program (if the collection is done "right") 5-15%.
- Considered a major factor determining program efficiency.
- Techniques have evolved since the 60's
- We will only go over basic techniques.

How to evaluate GC

- Throughput: Overall collection overheads.
- Responsiveness: Pauses in program run.
- Space overhead.
- Cache Locality (efficiency and energy).

Performance

- Responsiveness: refers to how quickly an application or system responds with a requested piece of data.
  - How quickly a desktop UI responds to an event
  - How fast a website returns a page
  - How fast a database query is returned
  - For applications that focus on responsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.
- Throughput: focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include:
  - The number of transactions completed in a given time.
  - The number of jobs that a batch program can complete in an hour.
  - The number of database queries that can be completed in an hour.
  - High pause times are acceptable for applications that focus on throughput. Since high throughput applications focus on benchmarks over longer periods of time, quick response time is not a consideration.

Three classic GC algorithms

- Reference counting
- Mark and sweep (and mark-compact)
- Copying

The last two are also called tracing algorithms because they go over (trace) all reachable objects.

Outline

Reference counting (Collins 1960)

Reference Counting Algorithm

Each object has an RC field;
new objects get o.RC:=1 ;
if *p that points to o1 is modified to point to o2* **then**
  o1.RC–;
  o2.RC++. ;
end
if *o1.RC==0* **then**
  Delete o1;
  Decrement o.RC for all children of o1 ;
  Recursively delete objects whose RC is decremented to 0 ;
end

```
1  Account x, y;
2  x = new SavingsAccount("1234567");
3  y = x;
4  x = null;
5  y = x;
```

rc for the object: $1 \rightarrow 2 \rightarrow 1 \rightarrow 0$

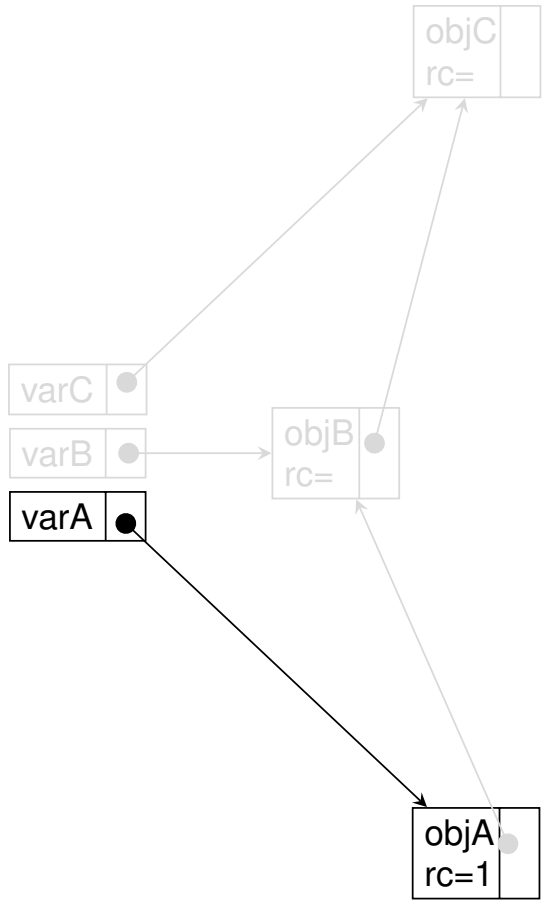
Reference counting

- Recall that we would like to know if an object is reachable from the roots.
- Associate a reference count field with each object: how many pointers reference this object.
- When nothing points to an object, it can be deleted.
- Very simple, used in many systems.

```
class LinkedList {
    LinkedList next;
}
int main() {
    LinkedList A = new LinkedList;
    LinkedList B = new LinkedList;
    LinkedList C = new LinkedList;
    A.next = B;
    B.next = C;
    B = C = null;
    A.next.next = null;
    A = null;
}
```

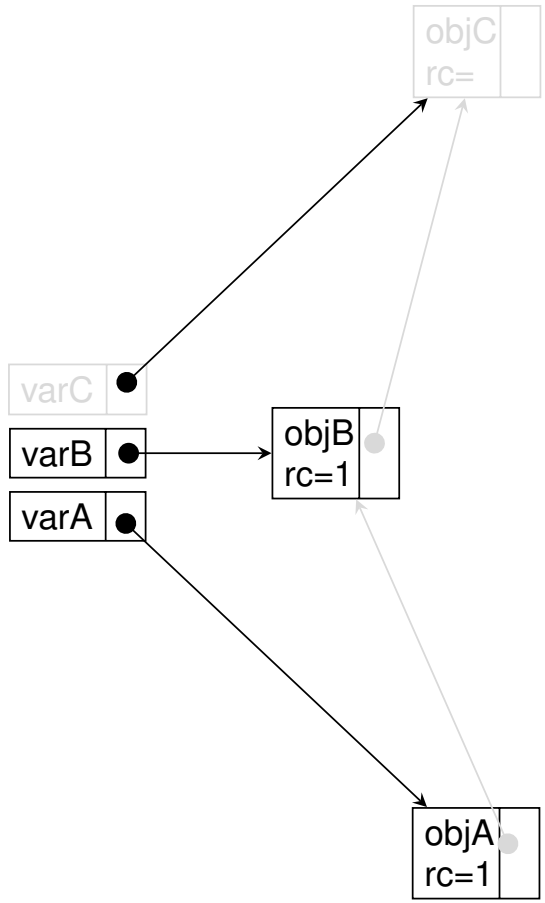
Tracing the Reference-count algorithm

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

B = C = null;

A.next.next = null;

A = null;
```



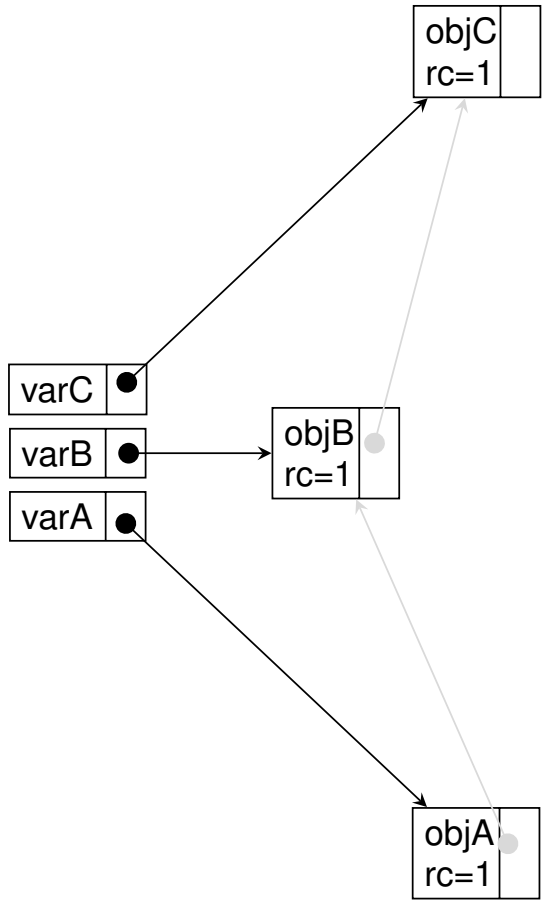
Tracing the Reference-count algorithm

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

B = C = null;

A.next.next = null;

A = null;
```



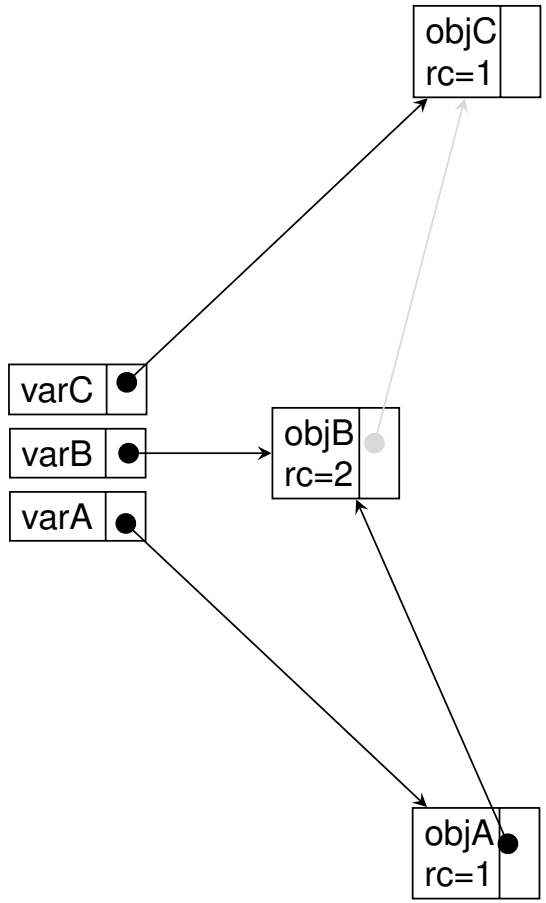
Tracing the Reference-count algorithm

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

B = C = null;

A.next.next = null;

A = null;
```



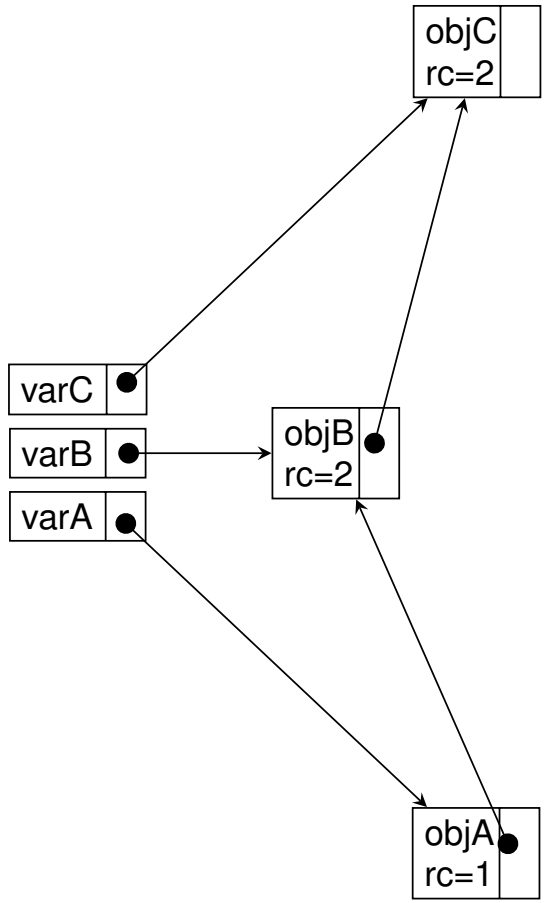
Tracing the Reference-count algorithm

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

B = C = null;

A.next.next = null;

A = null;
```



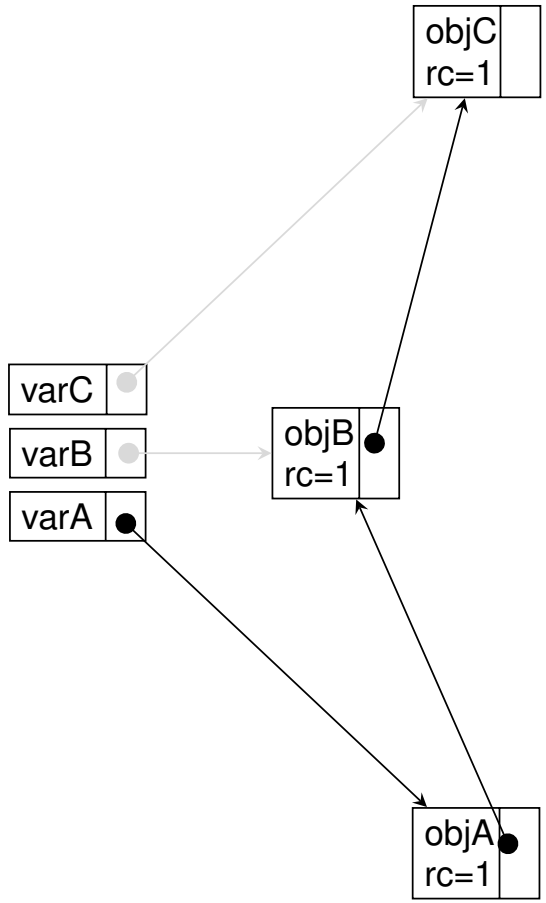
Tracing the Reference-count algorithm

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

B = C = null;

A.next.next = null;

A = null;
```



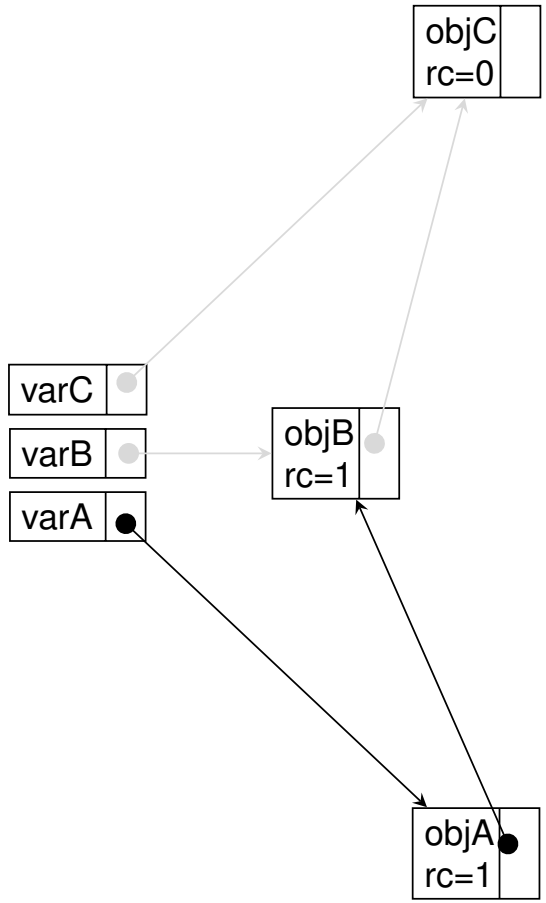
Tracing the Reference-count algorithm

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

B = C = null;

A.next.next = null;

A = null;
```



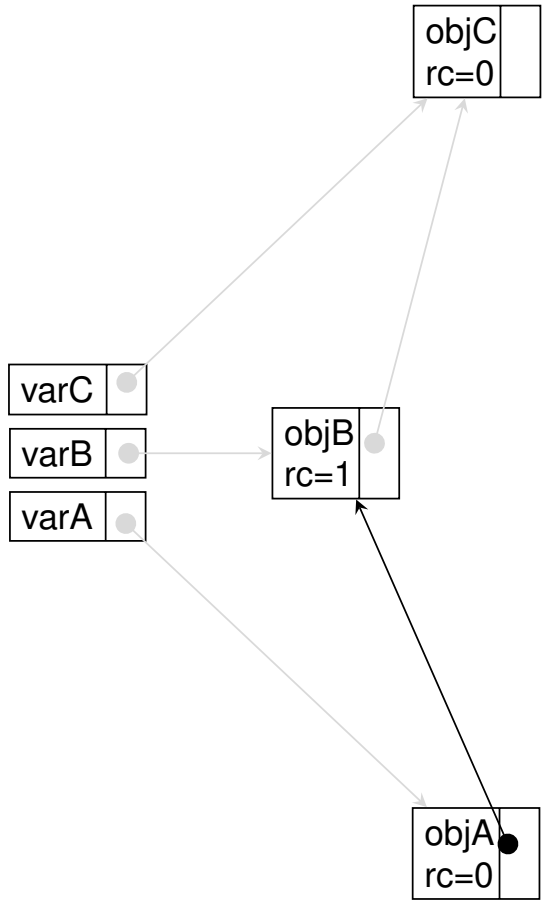
Tracing the Reference-count algorithm

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

B = C = null;

A.next.next = null;

A = null;
```



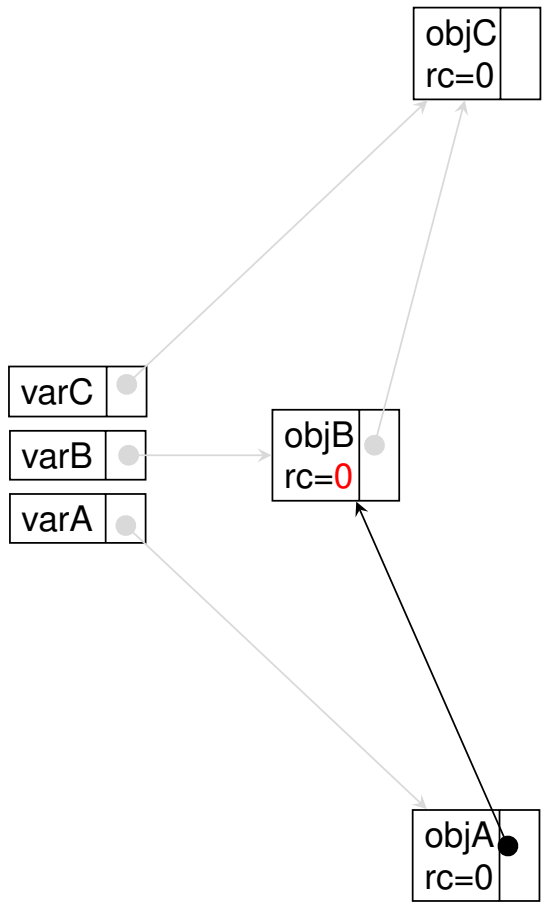
Tracing the Reference-count algorithm

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

B = C = null;

A.next.next = null;

A = null;
```



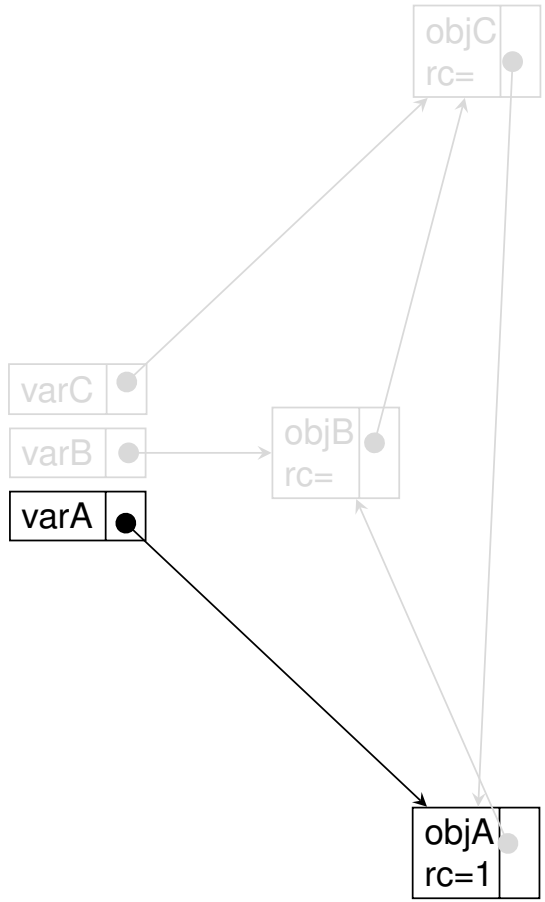
Tracing the Reference-count algorithm

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

B = C = null;

A.next.next = null;

A = null;
```



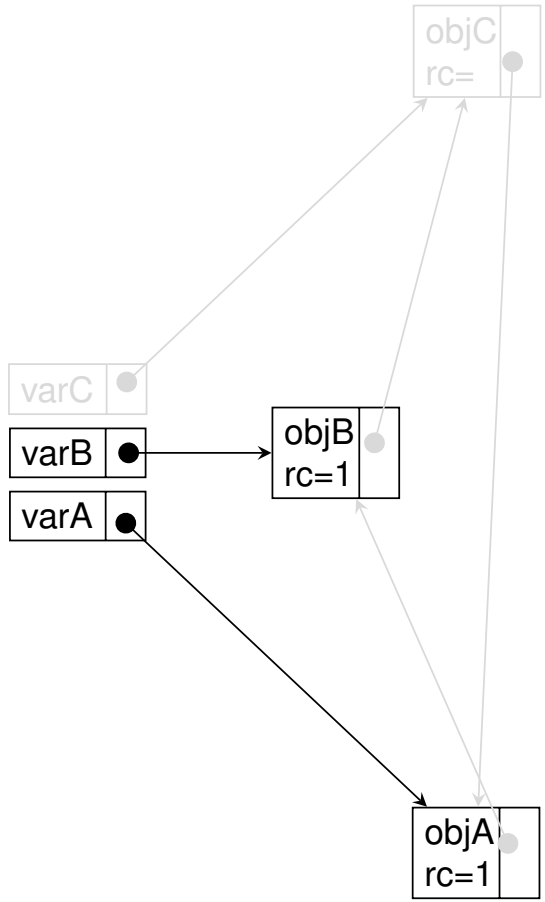
The problem of Reference-counting

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

C.next = A;

A = null;

B= null;

C = null;
```



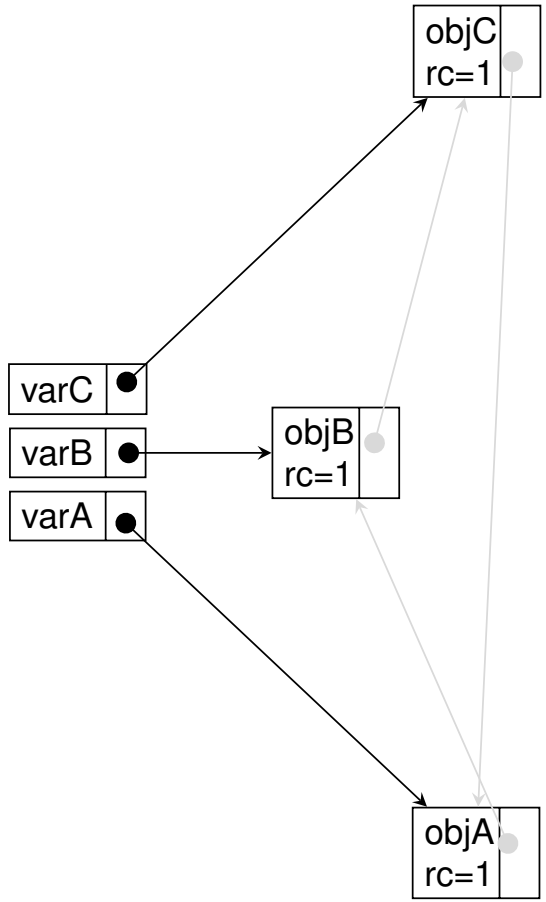
The problem of Reference-counting

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

C.next = A;

A = null;

B= null;

C = null;
```



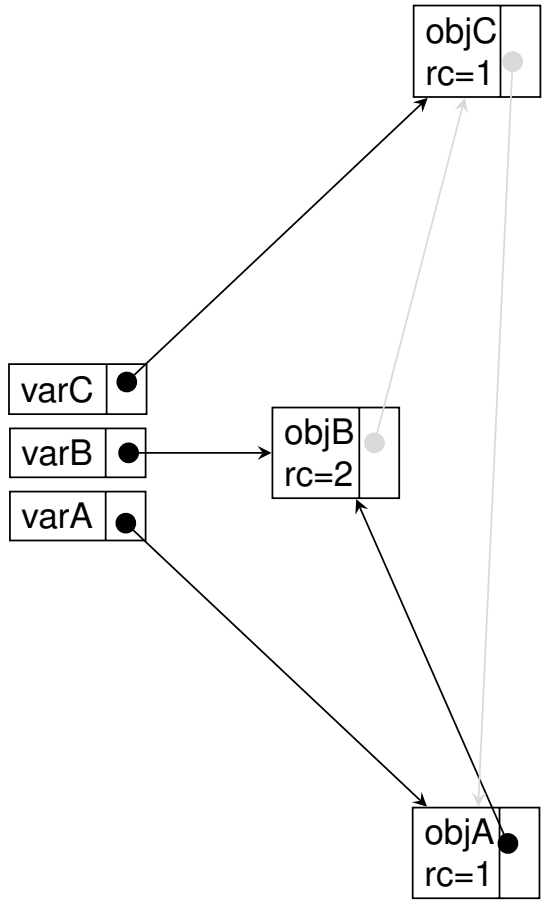
The problem of Reference-counting

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

C.next = A;

A = null;

B= null;

C = null;
```



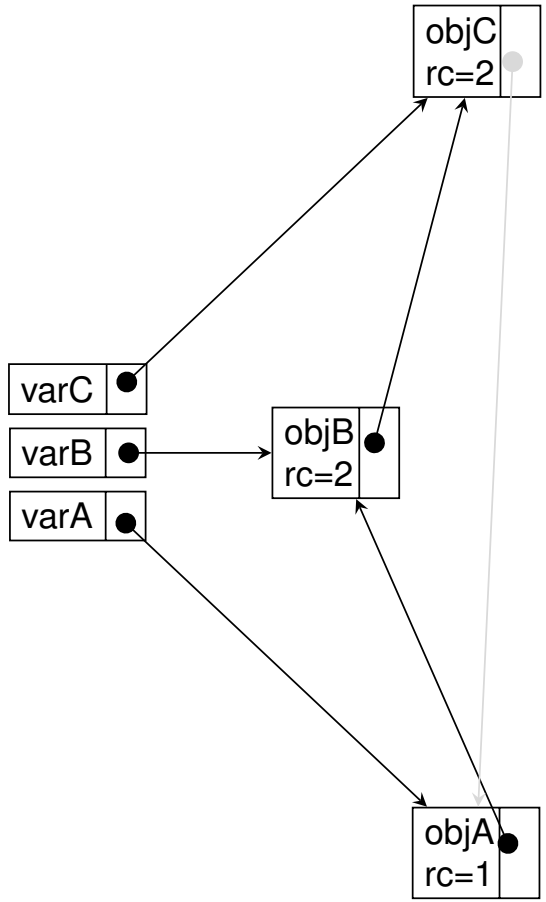
The problem of Reference-counting

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

C.next = A;

A = null;

B= null;

C = null;
```



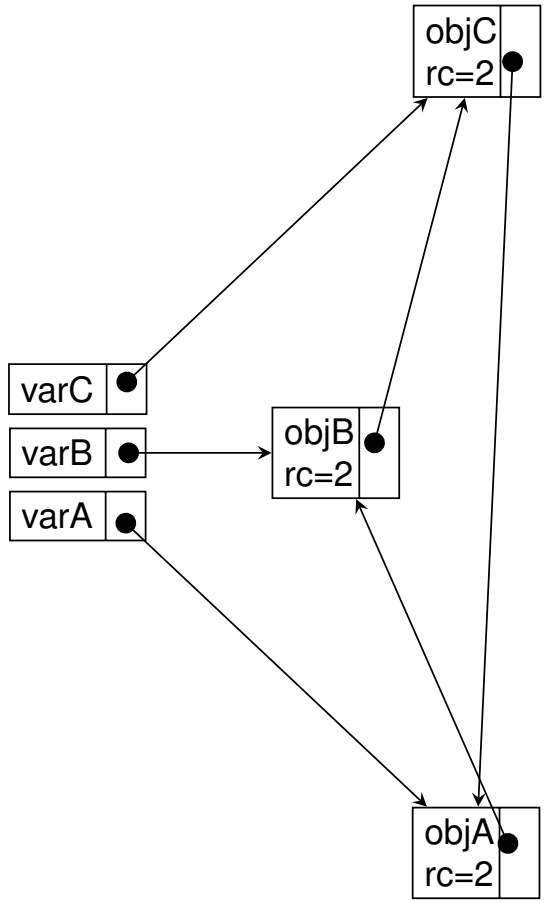
The problem of Reference-counting

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

C.next = A;

A = null;

B= null;

C = null;
```



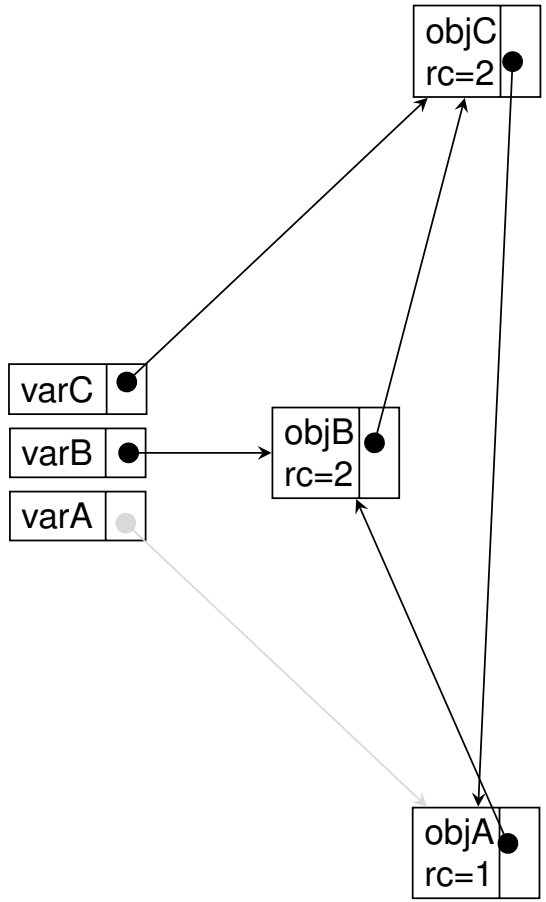
The problem of Reference-counting

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

C.next = A;

A = null;

B= null;

C = null;
```



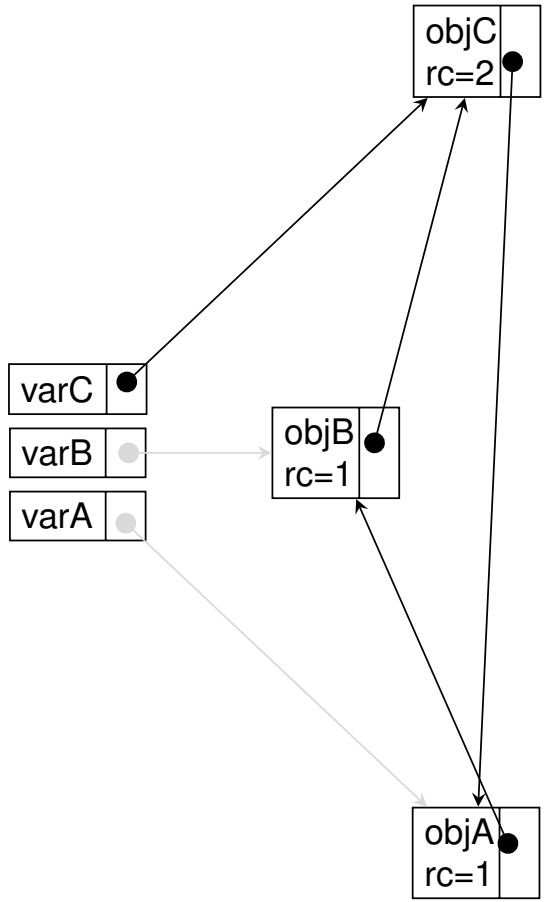
The problem of Reference-counting

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

C.next = A;

A = null;

B= null;

C = null;
```



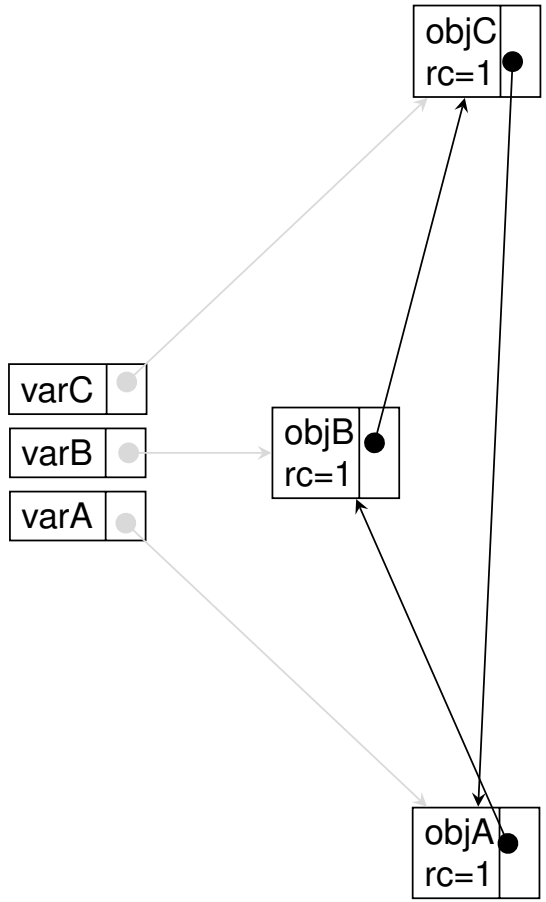
The problem of Reference-counting

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

C.next = A;

A = null;

B= null;

C = null;
```



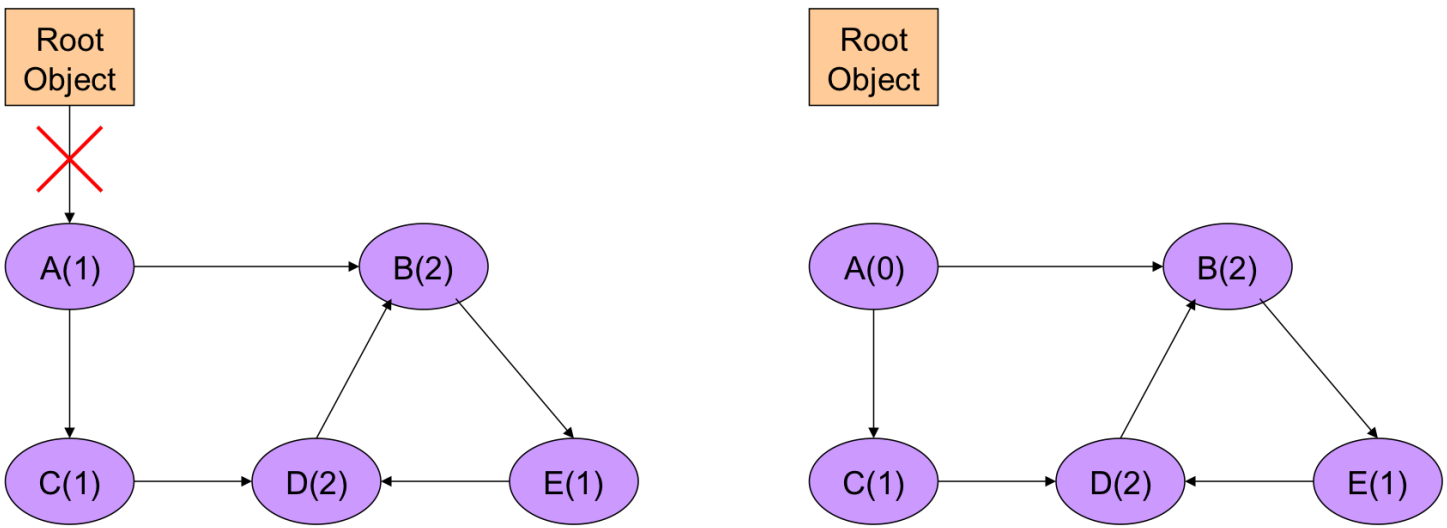
The problem of Reference-counting

```
LinkedList A = new LinkedList;

LinkedList B = new LinkedList;

LinkedList C = new LinkedList;

A.next = B;

B.next = C;

C.next = A;

A = null;

B= null;

C = null;
```



A problem for cycles

- The Reference counting algorithm does not reclaim cycles.
- Solution 1: ignore cycles, they do not appear frequently in modern programs;
- Solution 2: run tracing algorithms (that can reclaim cycles) infrequently;
- Solution 3: designated algorithms for cycle collection;



- Other algorithms?
- Observation: rather than explicitly keep track of the number of references to each object we can traverse all reachable objects and discard unreachable objects

Outline

(Naive) Mark-and-Sweep Algorithm [McCarthy 1960]

Mark phase:

- Start from roots and traverse all objects reachable by a path of pointers.
- Mark all traversed objects.

Sweep phase:

- Go over all objects in the heap.
- Reclaim objects that are not marked.

(animated figure from Wikipedia)

When GC starts

Garbage collection is triggered by allocation.

Triggering

Algorithm 1: New(A)

```
if freeList is empty then
    markSweep() ;
    if freeList is empty then
        return (“out-of-memory")
    end
    pointer = allocate(A);
    return (pointer);
end
```

- ►

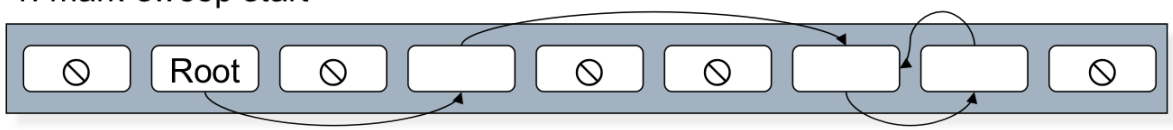
Properties of Mark-and-Sweep

Complexity:

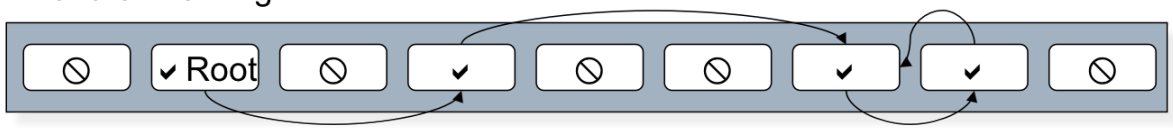
- Mark phase: live objects
- Sweep phase: heap size.
- Termination: each pointer traversed once.

- Most popular method today (at a more advanced form). Simple.
- Does not move objects, and so heap may fragment.
- Various engineering tricks are used to improve performance.

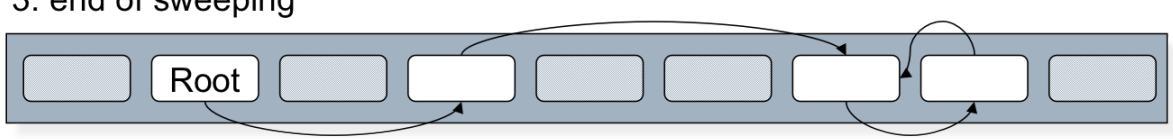
1. mark-sweep start



2. end of marking

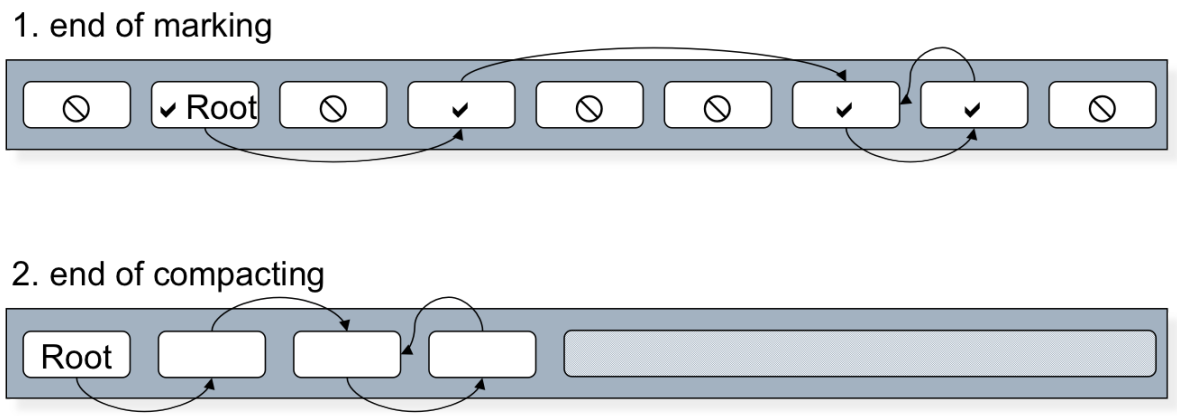


3. end of sweeping



Mark and compact

- During the run objects are allocated and reclaimed.
- Gradually, the heap gets fragmented.
- When space is too fragmented to allocate, a compaction algorithm is used.
- Move all live objects to the beginning of the heap and update all pointers to reference the new locations.
- Compaction is considered very costly and we usually attempt to run it infrequently, or only partially.



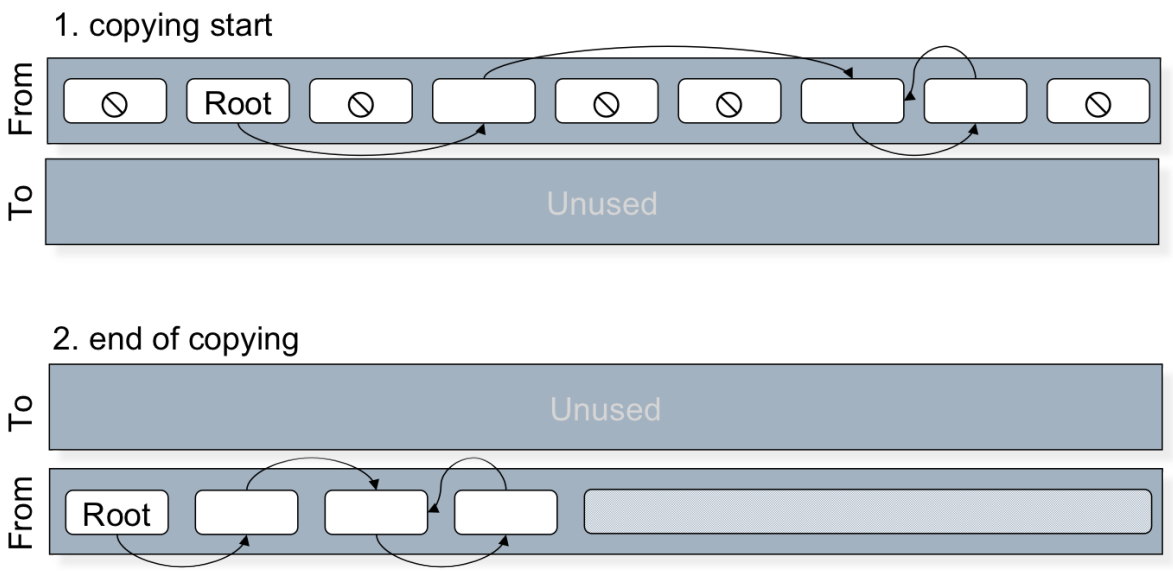
Compacting is a costly operation

Fragmented blocks of data VS compacted

Outline

Copying

- Heap partitioned into two.
- Part 1 takes all allocations.
- Part 2 is reserved.
- During GC, the collector traces all reachable objects and copies them to the reserved part.
- After copying the parts roles are reversed:
  - Allocation activity goes to part 2, which was previously reserved.
  - Part 1, which was active, is reserved till next collection.



Properties of Copying

- Compaction for free
- Major disadvantage: half of the heap is not used.
- “Touch" only the live objects
- Good when most objects are dead.
- Usually most new objects are dead, and so there are methods that use a small space for young objects and collect this space using copying garbage collection.

copying vs. mark-sweep

- Copying is good when live space is small (time) and heap is small (space).
- A popular choice: Copying for the (small) young generation. Mark-and-sweep for the full collection.
- A small waste in space, high efficiency.

Comparison

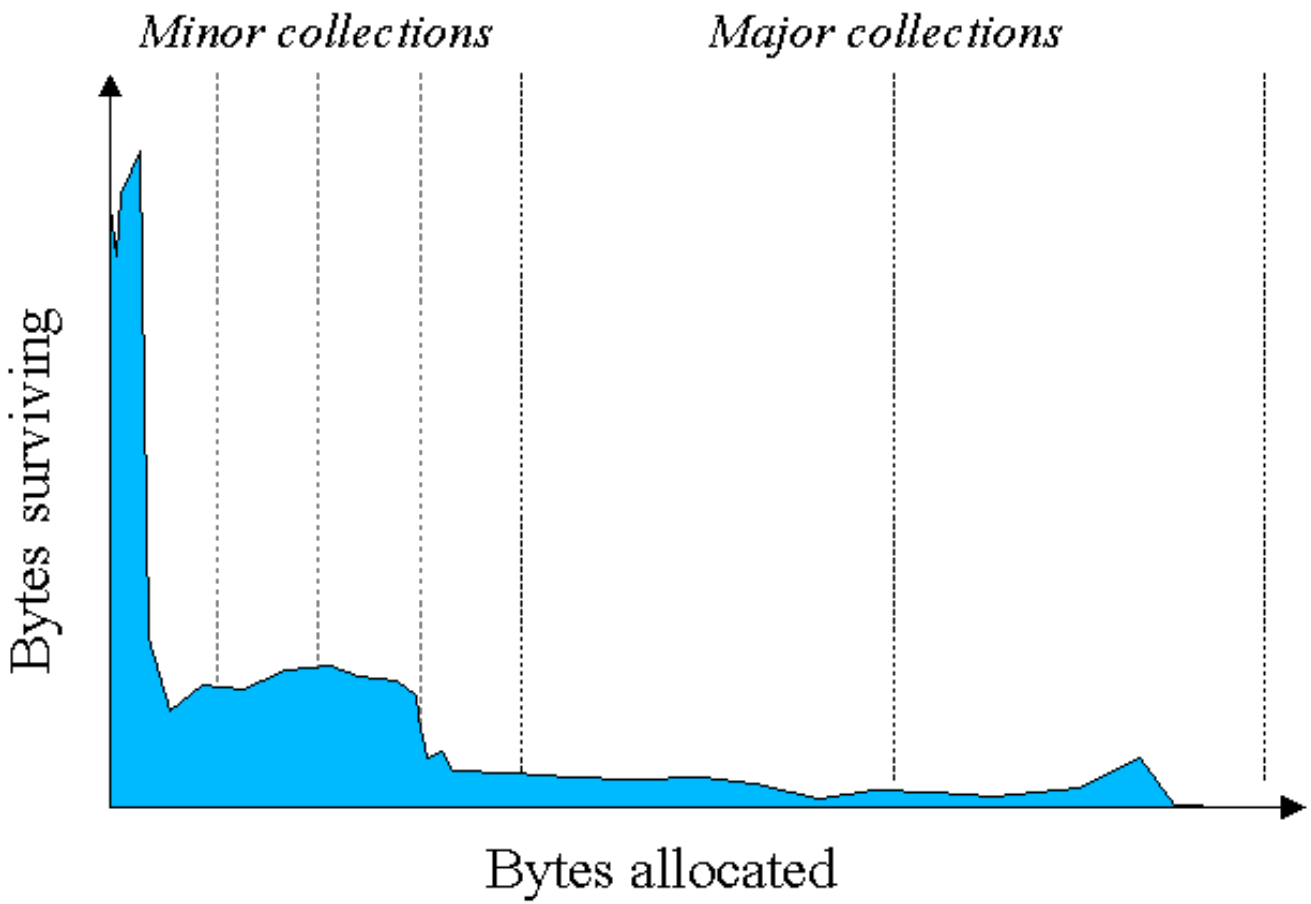
	Reference Counting	Mark & Sweep	Copying
Complexity	Pointer updates + dead objects	Size of heap (Live objects)	Live objects
Space overhead	Count/object	Bit/object	Half heap wasted
Compaction	Additional work	Additional work	For free
Pause time	Mostly short	long	Medium
More issues	Cycle collection		

Outline

Generational Garbage Collection

“The weak generational hypothesis":

most objects die young.



Generational GC

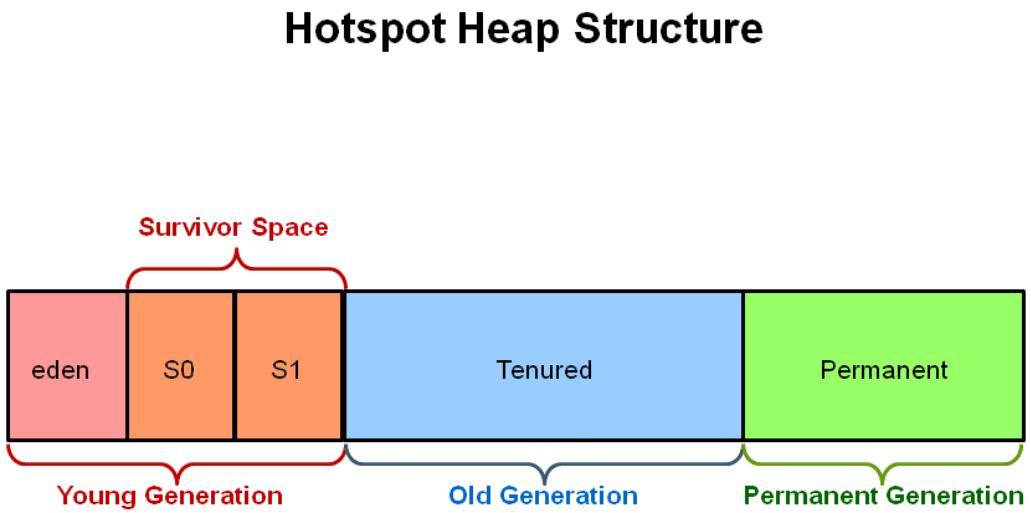
- Using the hypothesis: separate objects according to their ages and collect the area of the young objects more frequently.
- The heap is divided into two or more areas (generations).
- Objects allocated in 1st (youngest) generation.
- The youngest generation is collected frequently.
- Objects that survive in the young generation “long enough" are promoted to the old generation.

Advantages

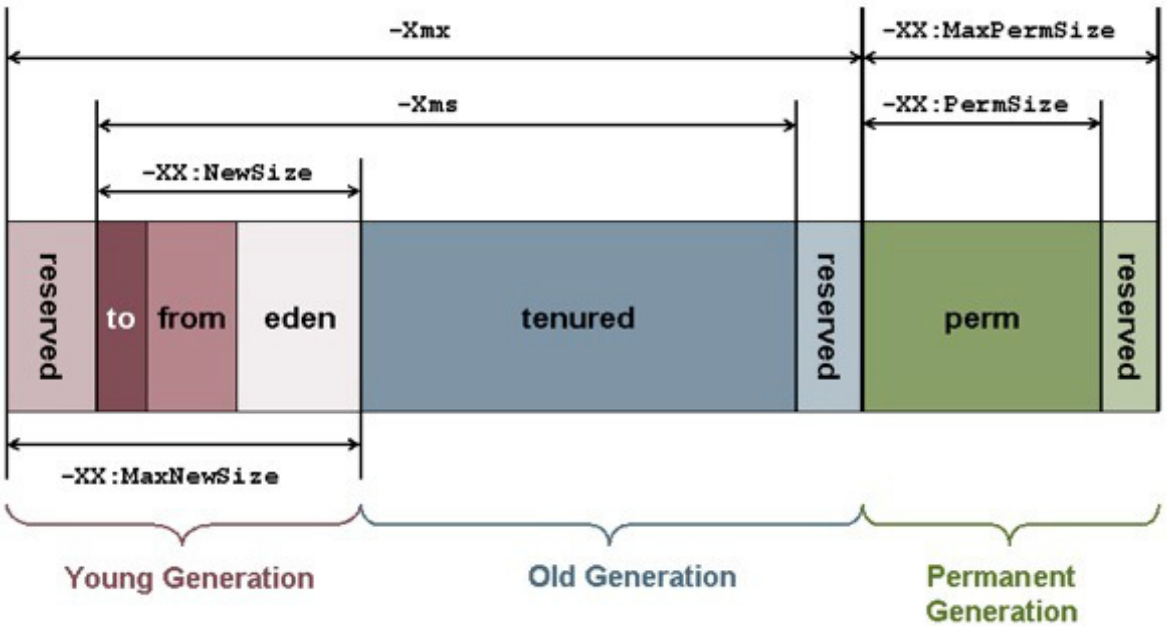
- Short pauses: the young generation is kept small and so most pauses are short.
- Efficiency: collection efforts are concentrated where many dead objects exists.
- Less fragments:
  - young and small objects might cause fragments. They are cleaned out in the young generation area.
  - old space becomes more compact
- Locality:
  - Collector: mostly concentrated on a small part of the heap;
  - Program: allocates (and mostly uses) young objects in a small part of the memory.

GC in Java

- New objects are allocated using a modified stop-and-copy collector in the Eden space.
- When Eden runs out of space, the stop-and-copy collector moves its elements to the survivor space.
- Objects that survive long enough in the survivor space become tenured and are moved to the tenured space.
- When memory fills up, a full garbage collection (perhaps mark-and-sweep) is used to garbage-collect the tenured objects.



Java GC Tuning



-Xms, -Xmx, -Xmn

```
public class gc{
public static void main(String[] args) throws Exception{
        int m=100000;
        int n=10000;
        long start = System.currentTimeMillis();
        Double [][] A=new Double [n][m];
        long end = System.currentTimeMillis();
        System.out.println(end-start);
```

- What heap memory we should request?
- -Xms is the minimal heap size

```
jianguos-MBP:code jianguolu$ java -Xms1g gc
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at gc.main(gc.java:7)
```

```
Jianguos-MBP:code jianguolu$ java -Xmn1g gc
15817
```

-Xms, -Xmx, -Xmn

```
public class gc{
public static void main(String[] args) throws Exception{
        int m=100000;
        int n=10000;
        long start = System.currentTimeMillis();
        Double [][] A=new Double [n][m];
        long end = System.currentTimeMillis();
        System.out.println(end-start);
```

- What heap memory we should request?
- -Xms is the minimal heap size

```
jianguos-MBP:code jianguolu$ java -Xms1g gc
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at gc.main(gc.java:7)
```

```
Jianguos-MBP:code jianguolu$ java -Xmn1g gc
15817
```

-Xms, -Xmx, -Xmn

```
public class gc{
public static void main(String[] args) throws Exception{
        int m=100000;
        int n=10000;
        long start = System.currentTimeMillis();
        Double [][] A=new Double [n][m];
        long end = System.currentTimeMillis();
        System.out.println(end-start);
```

- What heap memory we should request?
- -Xms is the minimal heap size

```
jianguos-MBP:code jianguolu$ java -Xms1g gc
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at gc.main(gc.java:7)
```

```
Jianguos-MBP:code jianguolu$ java -Xmn1g gc
15817
```

Excessive GC Time

- Why OutOfMemoryError ?
- The parallel collector throws an OutOfMemoryError if too much time is being spent in garbage collection (GC)
- If more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, then an OutOfMemoryError is thrown.
- This feature is designed to prevent applications from running for an extended period of time while making little or no progress because the heap is too small.
- If necessary, this feature can be disabled by adding the option -XX:-UseGCOverheadLimit to the command line.

Outline

Concurrent GC

Problem: Long pauses disturb the user

- An important measure for the collection: length of pauses.
- average pause time for compacting 1GB live data: one second

- Can we just run the program while the collector runs on a different thread?
- Not so simple!
- The heap changes while we collect.
- For example, we look at an object B, but before we have a chance to mark its children, the program changes them.

Memory Management with Parallel Processors

- Multiple threads are used to speed up garbage collection.
- Processor cores run GC in parallel
- Normally uses Stop the world GC for younger generations
  - `-XX:-UseParallelGC`
  - Use parallel garbage collection, mostly for younger generations
  - `-XX:-UseParallelOldGC`
  - Use parallel garbage collection for the full collections.
  - Enabling this option automatically sets -XX:+UseParallelGC.
- Different from Concurrent GC.

Takeaways

- What is the run time management, stack and heap.
- How to manage the memory, automated GC
- Classic GC algorithms: Reference counting, Mark and Sweep, Copying.
  - Reference counting: cyclic reference problem.
  - Mark and Sweep: needs to stop the machine; entire heap needs to be visited; compacting is costly.
  - Copying: half of the heap is wasted.
- Generational GC
- GC in java