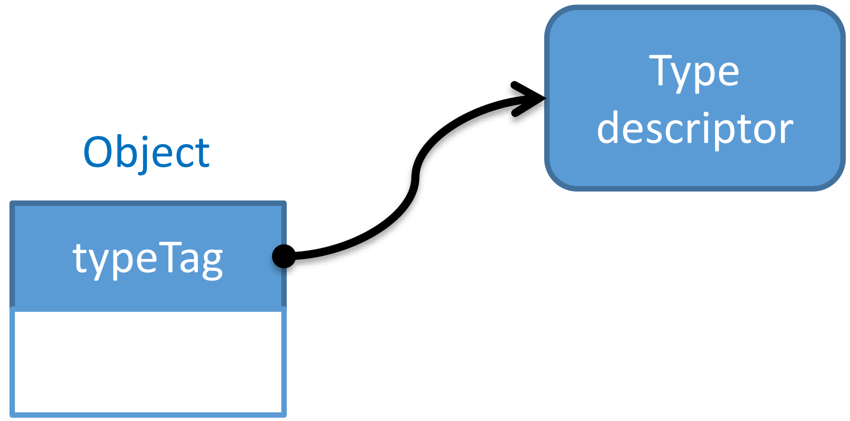
Course 142A Compilers & Interpreters

Garbage Collection

Lecture Week 8

Prof. Dr. Luc Bläser

Last Lecture - Quiz



What was the purpose of the type tag until now?

Type Tag: Purpose

Lookup type descriptor

- Ancestor table for type test and cast
- Virtual method table for dynamic dispatch
- Metadata for the interpreter (field/array types)

New:

- Pointer offsets for garbage collection

Today's Topics

- Memory Safety
- Garbage Collection
- Mark & Sweep
- GC Metadata

Learning Goals

- Understand the purpose and functionality of a Garbage Collector
- Know how to implement a simple Mark & Sweep GC for your runtime system

Memory Deallocation

- Metadata
  – No deallocation needed
- Stack
  – Return from method
- Heap
  – Object deallocation

Our focus

Do we need object deallocation at all?
If yes, how can we do this?

Explicit Deletion

- delete-statement to deallocate objects
  - Opposite of new-statement
  - Offered in e.g. C/C++/Pascal

```
x = new T();
...
delete x;
```

What is the problem with this?

Problems of Explicit Deletion

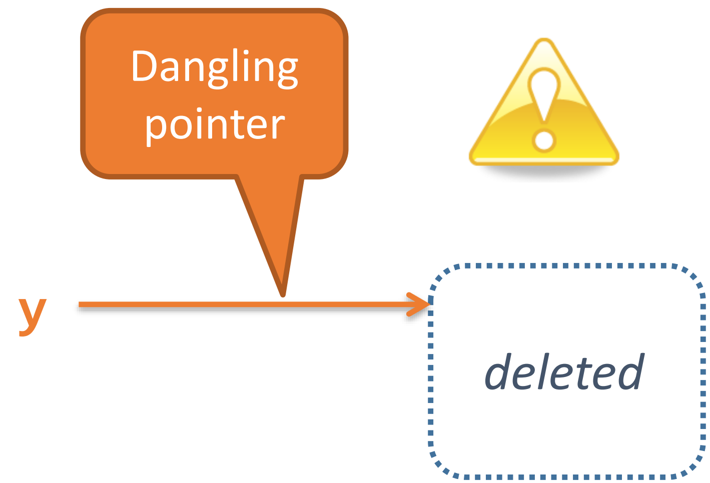
- Dangling pointers
  - Reference to an already deleted object
- Memory leaks
  - Orphan objects that cannot be removed

Serious memory errors

Dangling Pointer

- Reference to an object that has already been deleted

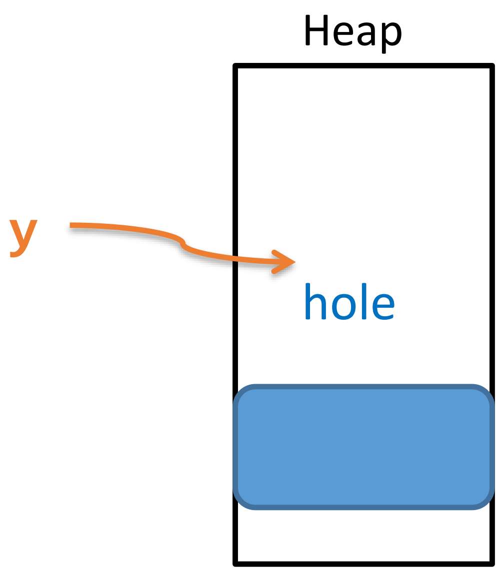
```
x = new T();
y = x;
...
delete x;
```

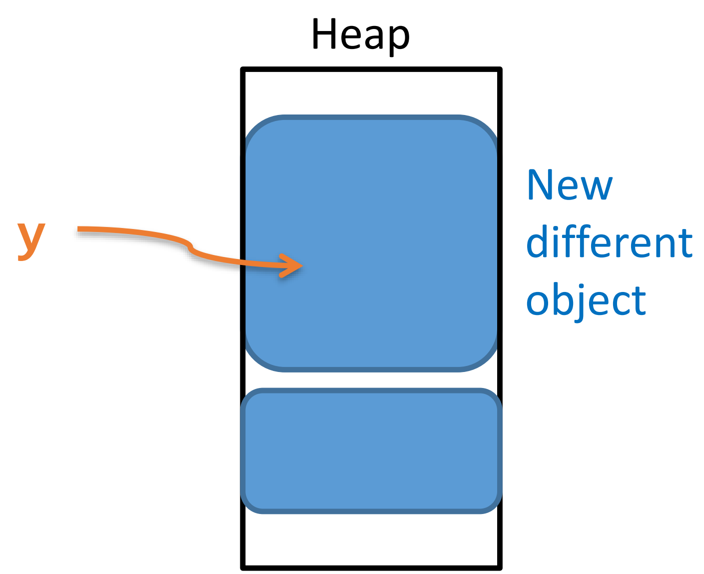


Why is this dangerous?

Dangling Pointer Problem

- Points to a hole or wrong object in heap
- Read unauthorized memory (security issue)
- Overwrite unrelated memory (safety/security issue)

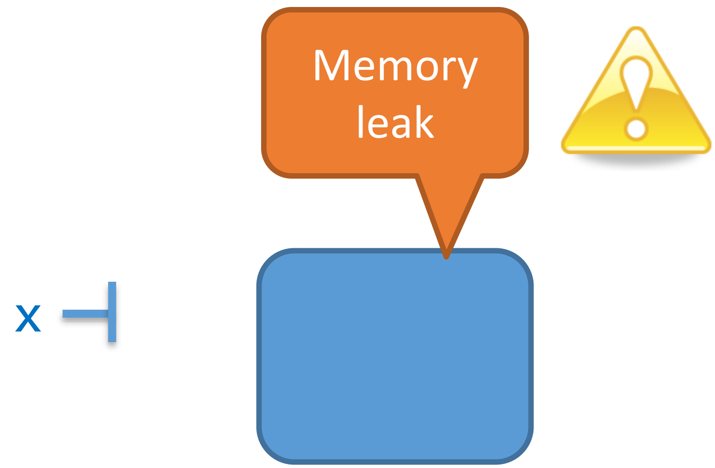




Memory Leak

- Unneeded object that is undeletable
- There exists no accessible reference to it

```
x = new T();
x = null;
...
```



Undeletable garbage fills the heap

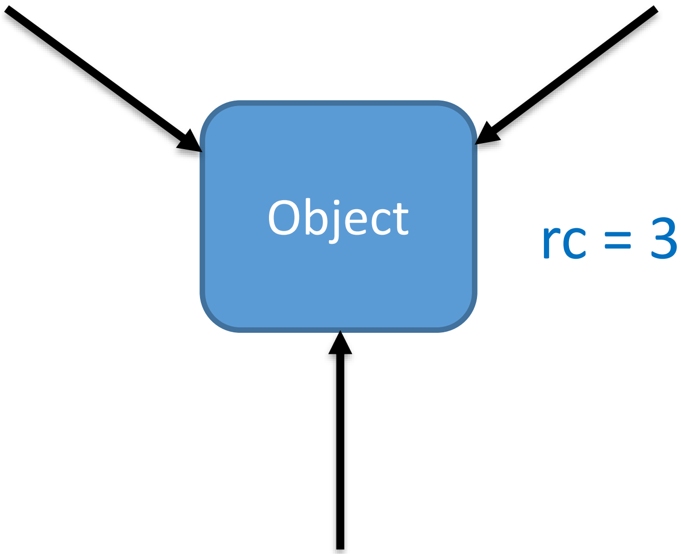
Garbage Collection

- Runtime system takes care of automatic reclamation of garbage
- Garbage = objects that are unreachable (and therefore no longer used)

Goal: Memory safety
- No dangling pointers
- No memory leaks

Reference Counting

- Reference counter rc per object
  - Number of incoming references



rc == 0 => Garbage

Inverse does not apply

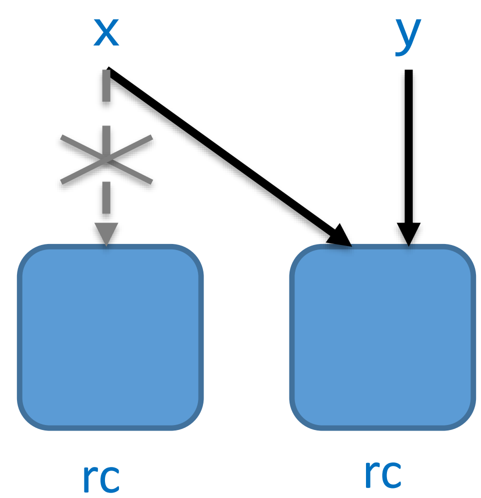
Expensive Updates

- On reference assignments (here both non-null)

```
x = y;
```

Actual code:

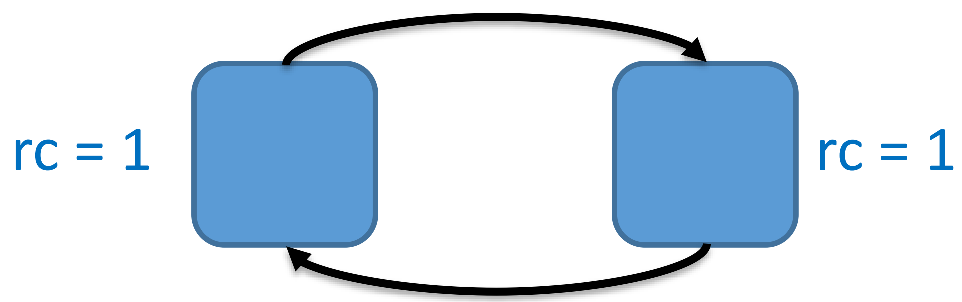
```
y.rc++;
x.rc--;
if (x.rc == 0) {
  delete x;
}
x = y;
```



Atomic increment/decrement
in the presence of concurrency

Cycles

- Cyclic object structures will never become garbage with reference counting



Memory leak

Reference Counting

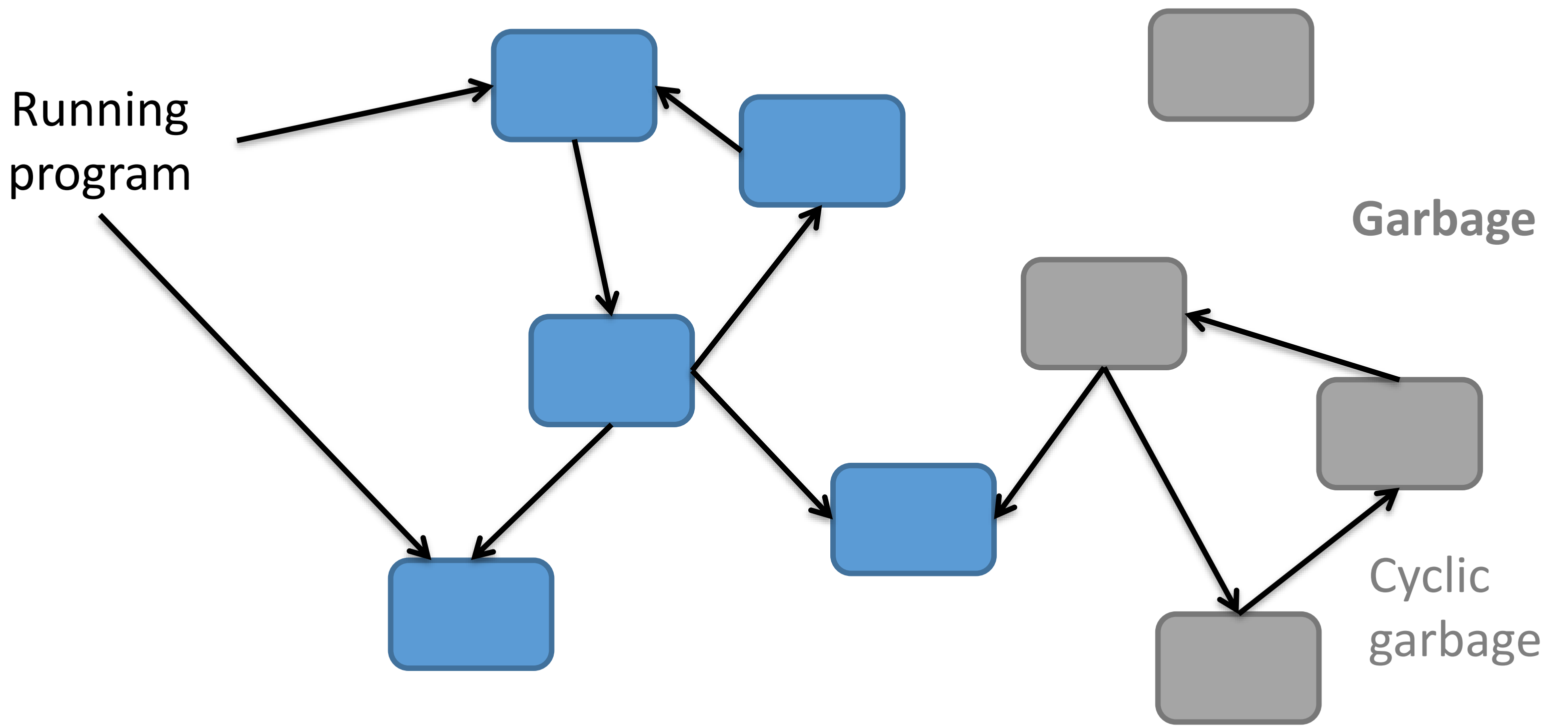
- Advantage
  - Immediate deallocation
- Disadvantage
  - Not suited for cycles
  - Slow

Unsuited

- Applied in C++ (smart pointers), Objective C, Swift
  - Provisional solution with weak pointers on cycles
  - Problem remain: Memory leaks and premature deletes
- Only suited for acyclic memory

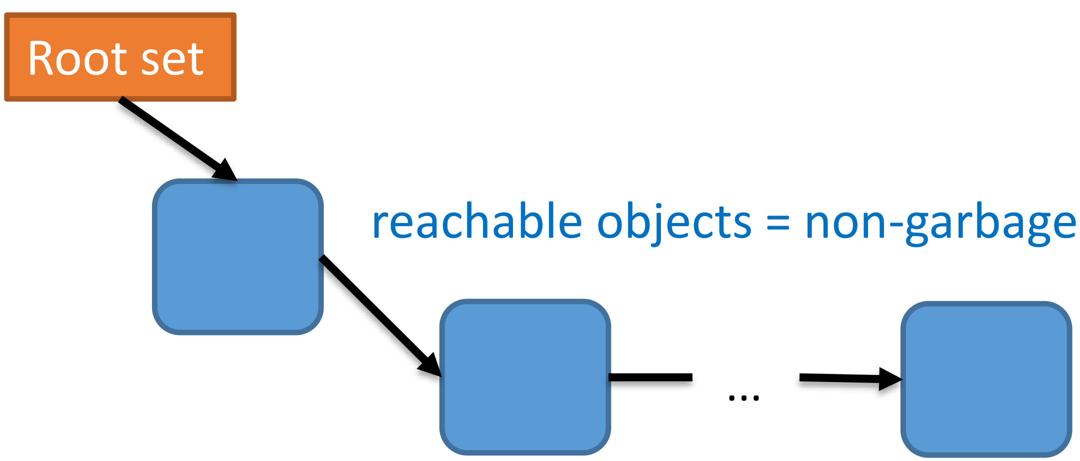
Garbage Collector (GC)

- Runtime system analyzes heap and deletes garbage
- Garbage = Unreachable objects from the program



Transitive Reachability

- Keep objects that could be accessed by the program in the future
- All directly or indirectly reachable objects via references from the program
- Starting from root set



Root Set

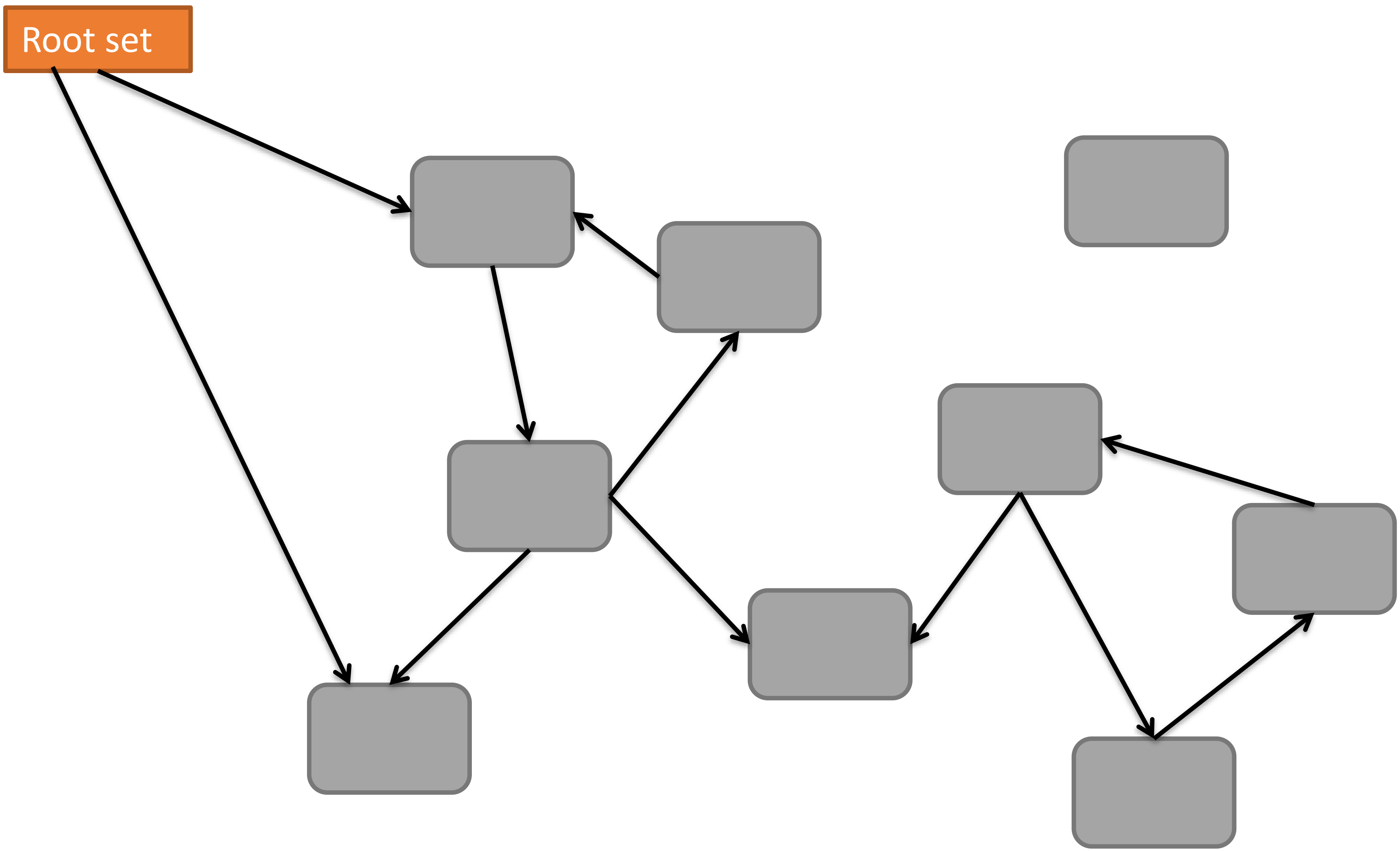
- References in static variables (if applicable)
- References on call stack (activation frames)
- References in registers (if applicable)

Mark & Sweep Algorithm

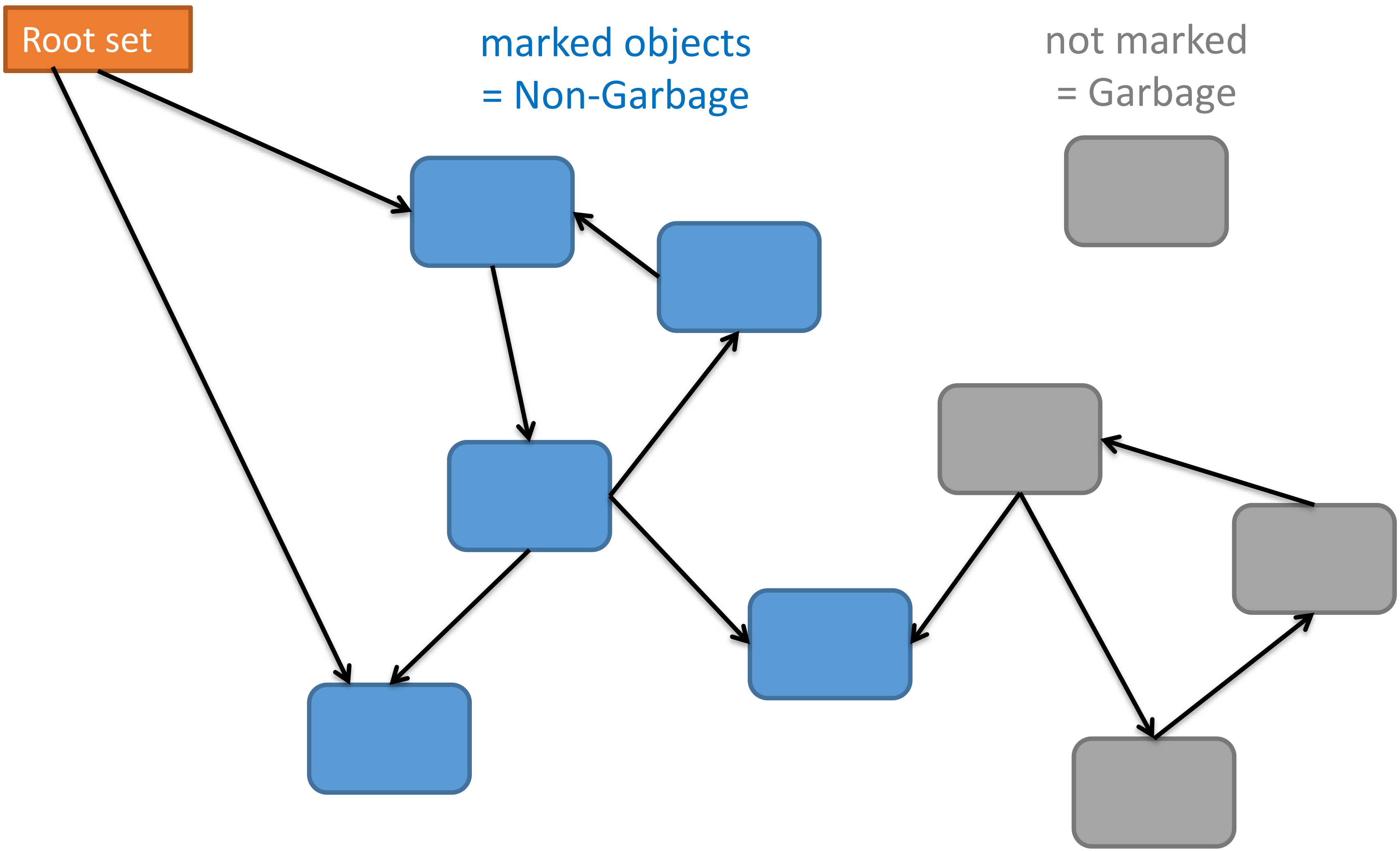
- Mark phase
  - Mark all reachable objects
- Sweep phase
  - Delete all unmarked objects

```
void collect() {
   mark();
   sweep();
}
```

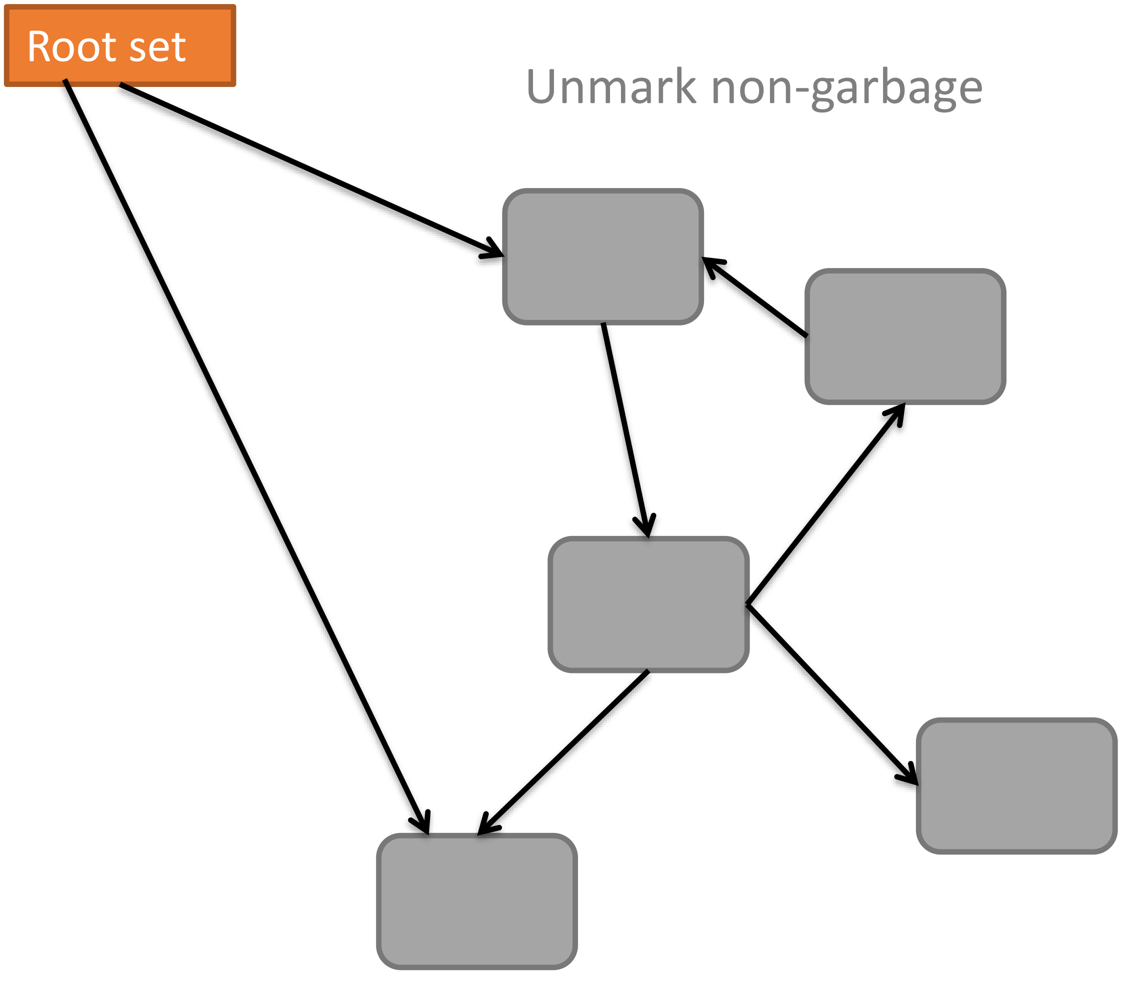
Example: GC



Example: Mark Phase



Example: Sweep Phase



Unmark non-garbage

Free garbage

Mark Phase: Implementation

Enumerable root set

```
void mark() {
  for (var root : getRootSet()) {
    traverse(root);
  }
}
```

traverse reachable objects

Depth-First Traversal

Mark flag in object layout

```
void traverse(Pointer current) {
  if (current != null && !isMarked(current)) {
    setMark(current);
    for (var next : getPointers(current)) {
      traverse(next);
    }
  }
}
```

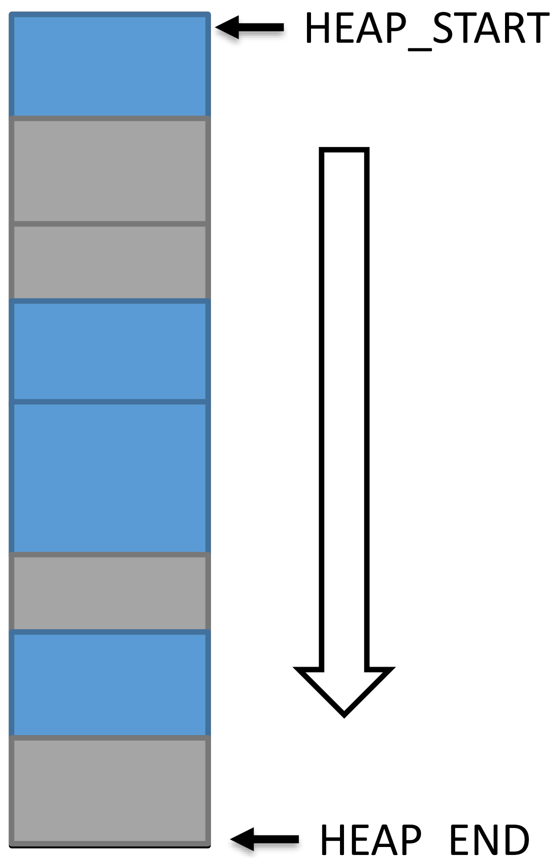
Enumerate references in object

Sweep: Approach

- Linear scan over entire heap, all blocks

marked object

unmarked block



Sweep: Implementation

```
void sweep() {
  var current = HEAP_START;
  while (current < HEAP_SIZE) {
     if (!isMarked(current)) {
       free(current);
     }
     clearMark(current);
     current += heap.getBlockSize(current);
  }
}
```

Register free block in heap

Detailed Aspects

What questions remain open?

Open Questions

General

- When does the GC run?
- Can the program run during GC?

Mark phase

- How to collect the root set?
- Where are the references inside an object?

Sweep phase

- How to determine the block size?
- Where to pass the free blocks?

Point of Execution

- Delayed garbage collection
  - Garbage is not immediately detected and freed
- GC runs at latest when the heap is full
  - Check in the allocate-method
- Possibly earlier for prophylactic reasons
  - In particular with finalizers (discussed next lecture)

Stop & Go

- GC runs sequentially and exclusively
- Mutator = Productive program
- Mutator is interrupted during GC

Time axis:

Other mechanism: Next lecture

Root Set Collection

Pointers on call stack

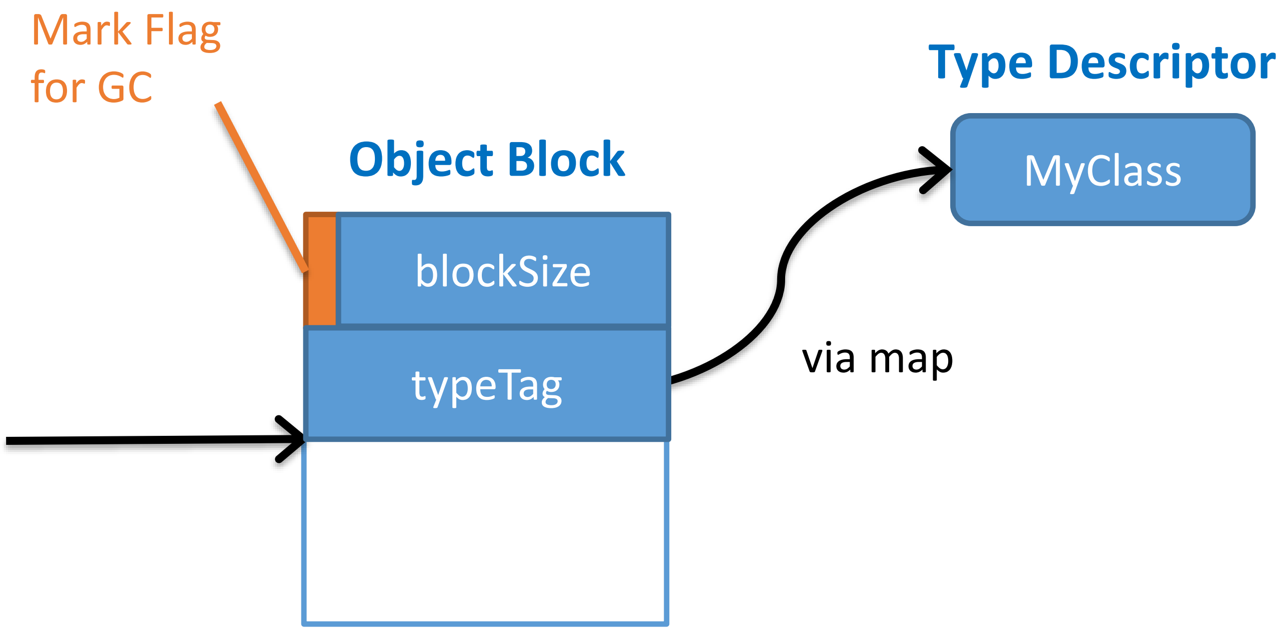
- Pointers in parameters
- Pointers in locals
- Pointers on evaluation stack
- “this”-reference

(no static fields or registers in our case)

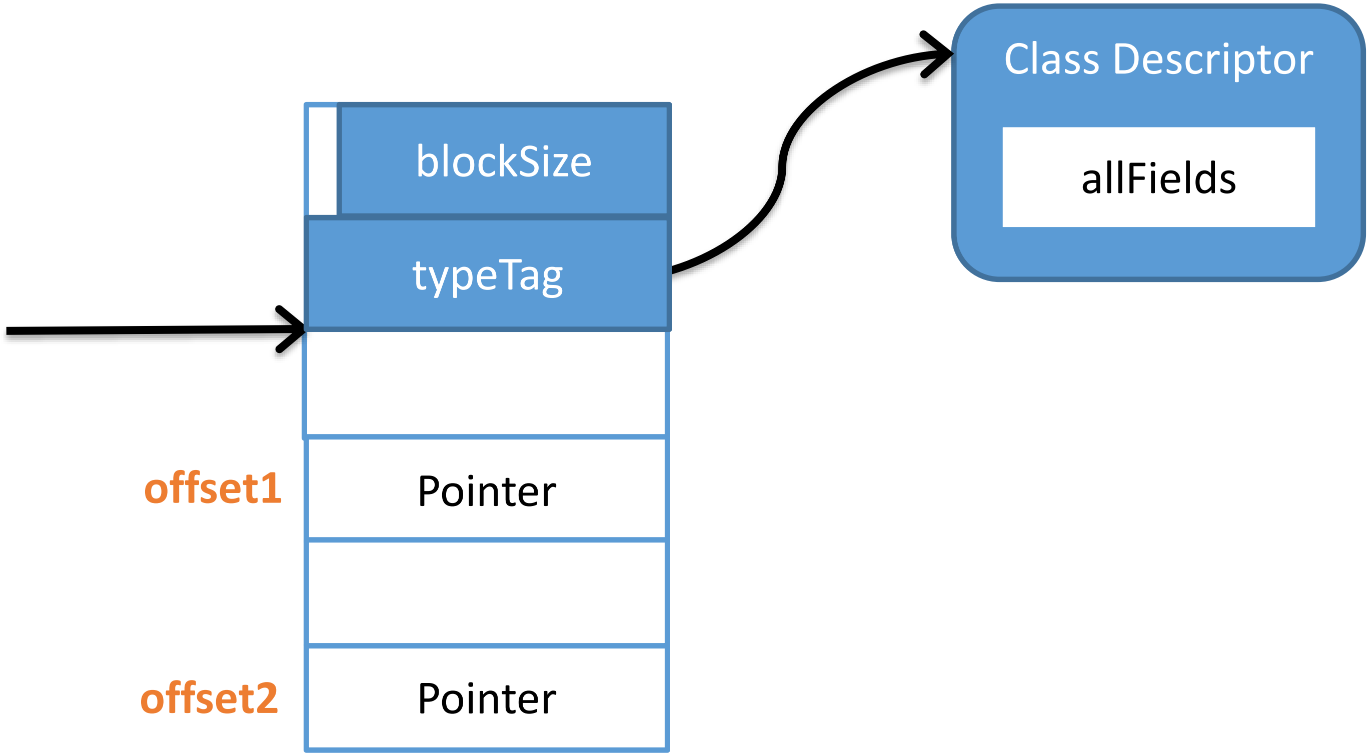
Root Set Collection

```
Iterable<Pointer> getRootSet(CallStack callStack) {
  var list = new ArrayList<Pointer>();
  for (var frame : callStack) {
    collectPointers(frame.getParameters());
    collectPointers(frame.getLocals());
    collectPointers(frame.getEvaluationStack().toArray());
    list.add(frame.getThisReference());
  }
  return list;
}
```

Mark Flag


Mark Flag
for GC
Object Block
blockSize
typeTag
via map
Type Descriptor
MyClass

Pointers in Object



How can we figure out the pointer offsets in the object?

Pointers in Object

```
Iterable<Pointer> getPointers(Pointer current) {
  var list = new ArrayList<Pointer>();
  var descriptor = heap.getDescriptor(current);
  var fields = ((ClassDescriptor)descriptor).getAllFields();
  for (var index = 0; index < fields.length; index++) {
    if (isPointerType(fields[index].getType())) {
      var value = heap.readField(current, index);
      if (value != null) {
        list.add((Pointer) value);
      }
    }
  }
  return list;
}
```

isPointerType() = class or array descriptor

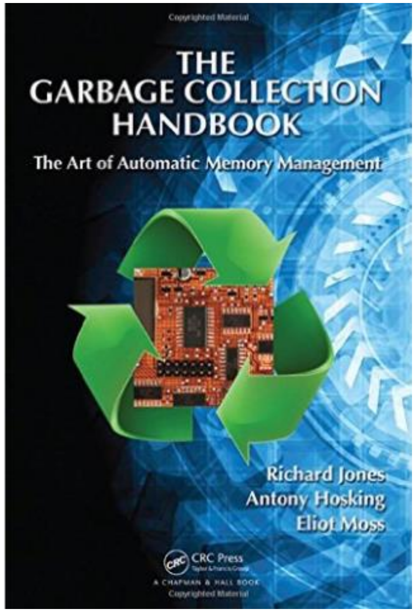
Consider arrays additionally!

Review: Learning Goals

- ✓ Understand the purpose and functionality of a Garbage Collector
- ✓ Know how to implement a simple Mark & Sweep GC for your runtime system

Further Reading

- Dragon Book, Garbage Collection
  - Section 7.5-7.6.2: Mark and sweep
- Optional, if interested
  - R. Jones, A. Hosking und E. Moss. The Garbage Collection Handbook. Chapman & Hall, 2011



Course 142A Compilers & Interpreters

Garbage Collection

Lecture Week 8, Wednesday

Prof. Dr. Luc Bläser

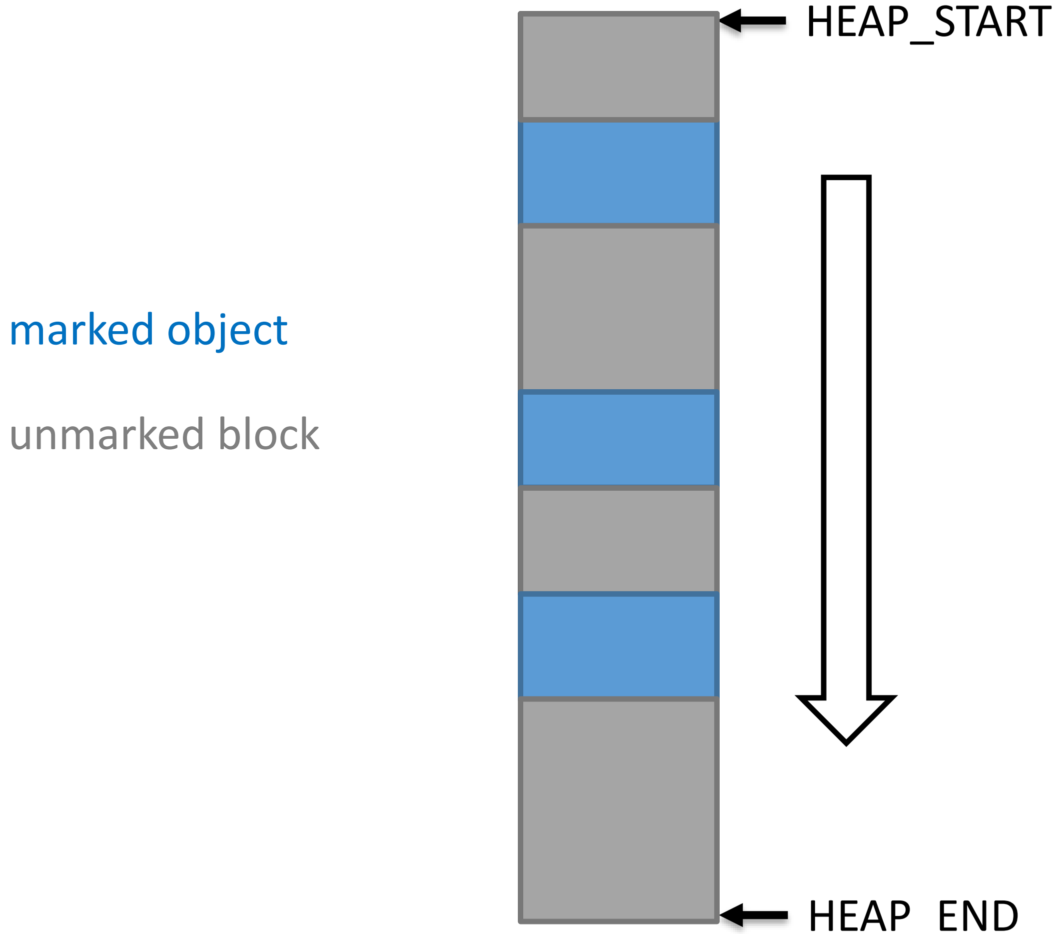
Last Lecture - Quiz

```
void sweep() {
  var current = HEAP_START;
  while (current < HEAP_SIZE) {
     if (!isMarked(current)) {
       free(current);
     }
     clearMark(current);
     current += heap.getBlockSize(current);
  }
}
```

What should we do with the free blocks?

Sweep

- Remember the free blocks for later re-allocation



Today's Topics

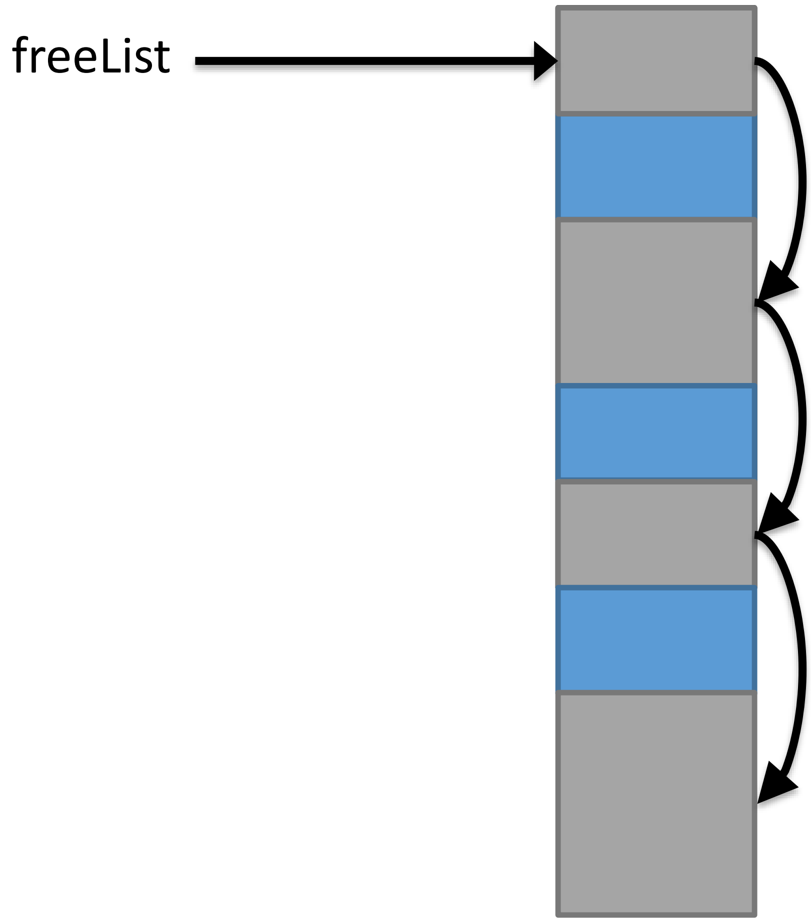
- Free Lists
- Advanced GC Topics

Learning Goals

- Understand how free heap blocks are managed
- Gain principal knowledge of advanced GC mechanisms

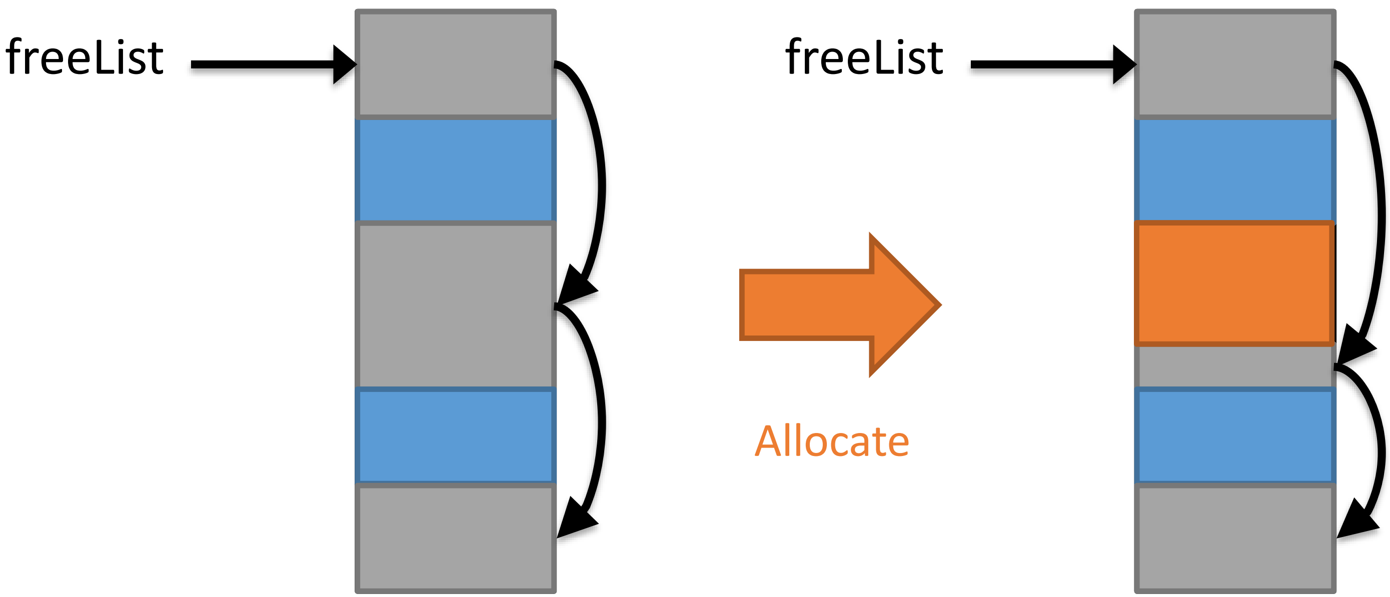
Free List

- Linearly linked list of free blocks

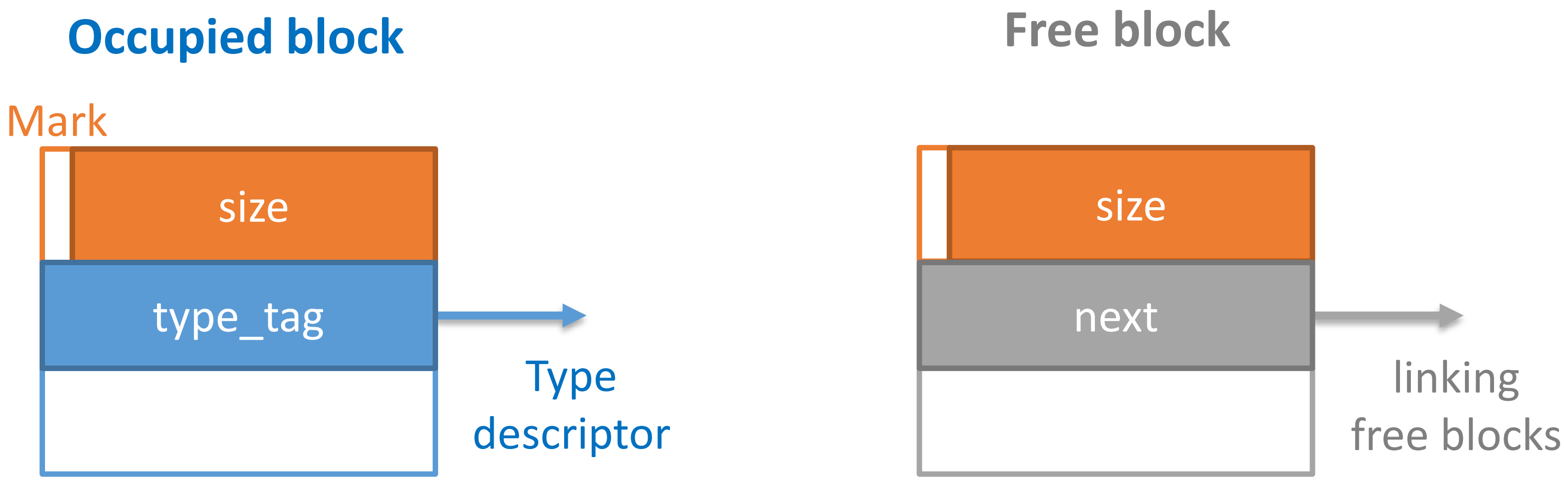


New Heap Allocation

- Traverse free list until a fitting block is found
- Left-over of block can be re-inserted in free list



Heap Block Layouts



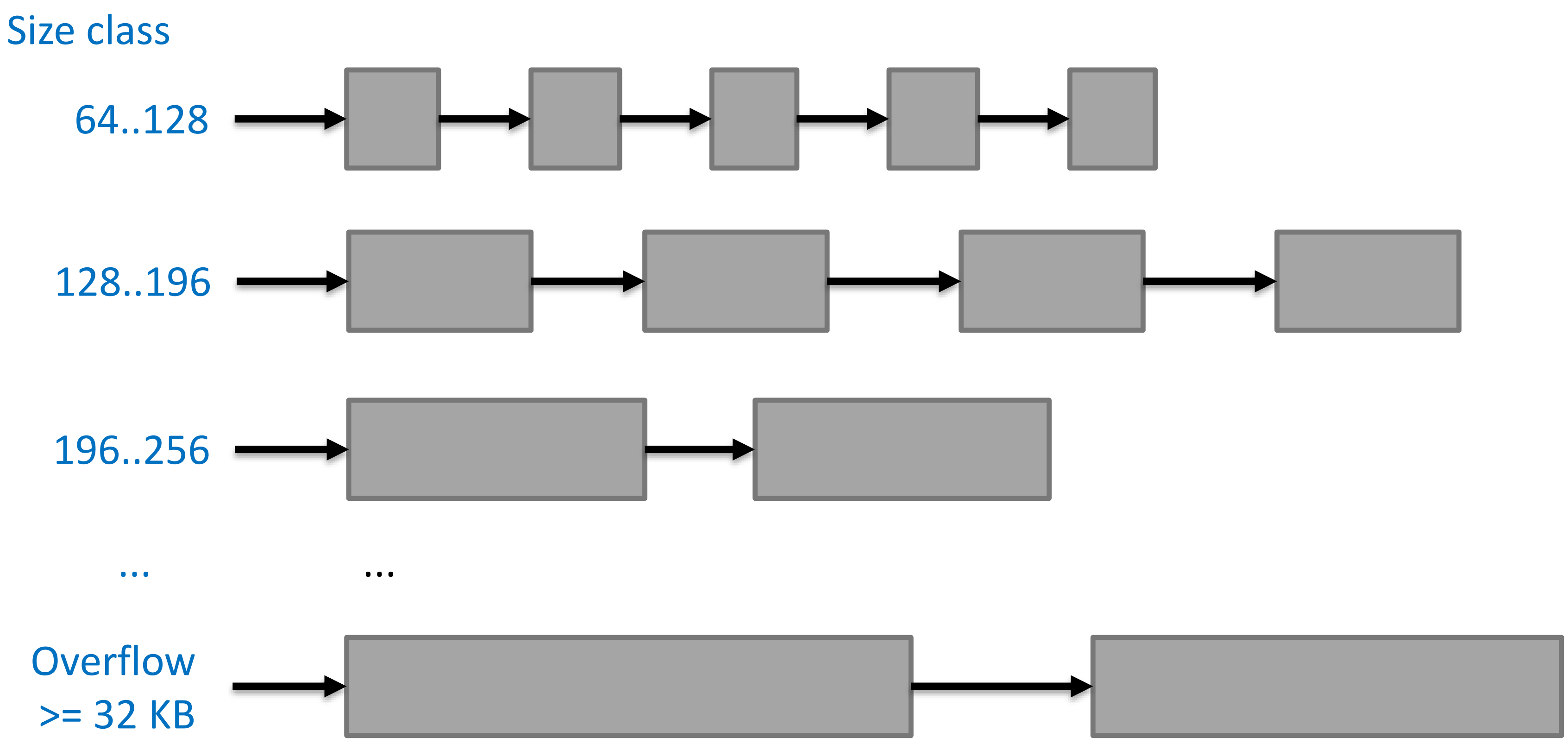
Sweep requires symmetric block header (mark/size)

Free List Strategies

- First Fit
  - No sorting
  - Search for first fitting block
- Best Fit
  - Ascending sorting by size
  - Useless small fragments
- Worst Fit
  - Descending sorting by size
  - Find fitting block immediately

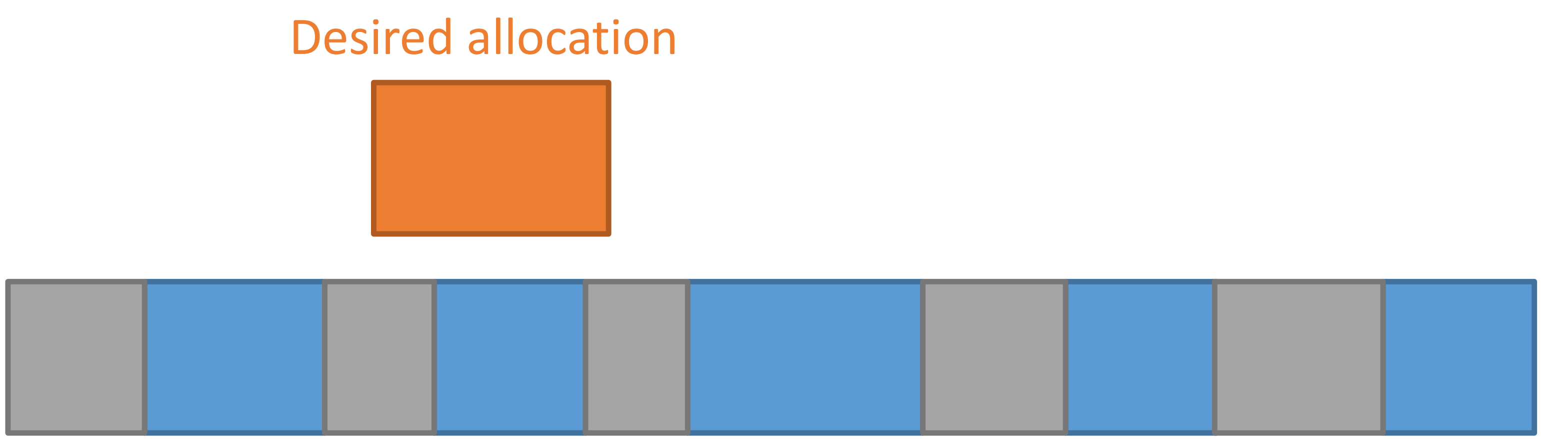
Segregated Free List

- Multiple free lists with different size classes



External Fragmentation

- Many small holes in heap due to allocate & free
  - Larger allocation may no longer fit into a hole
  - Although sum of free blocks would be sufficient



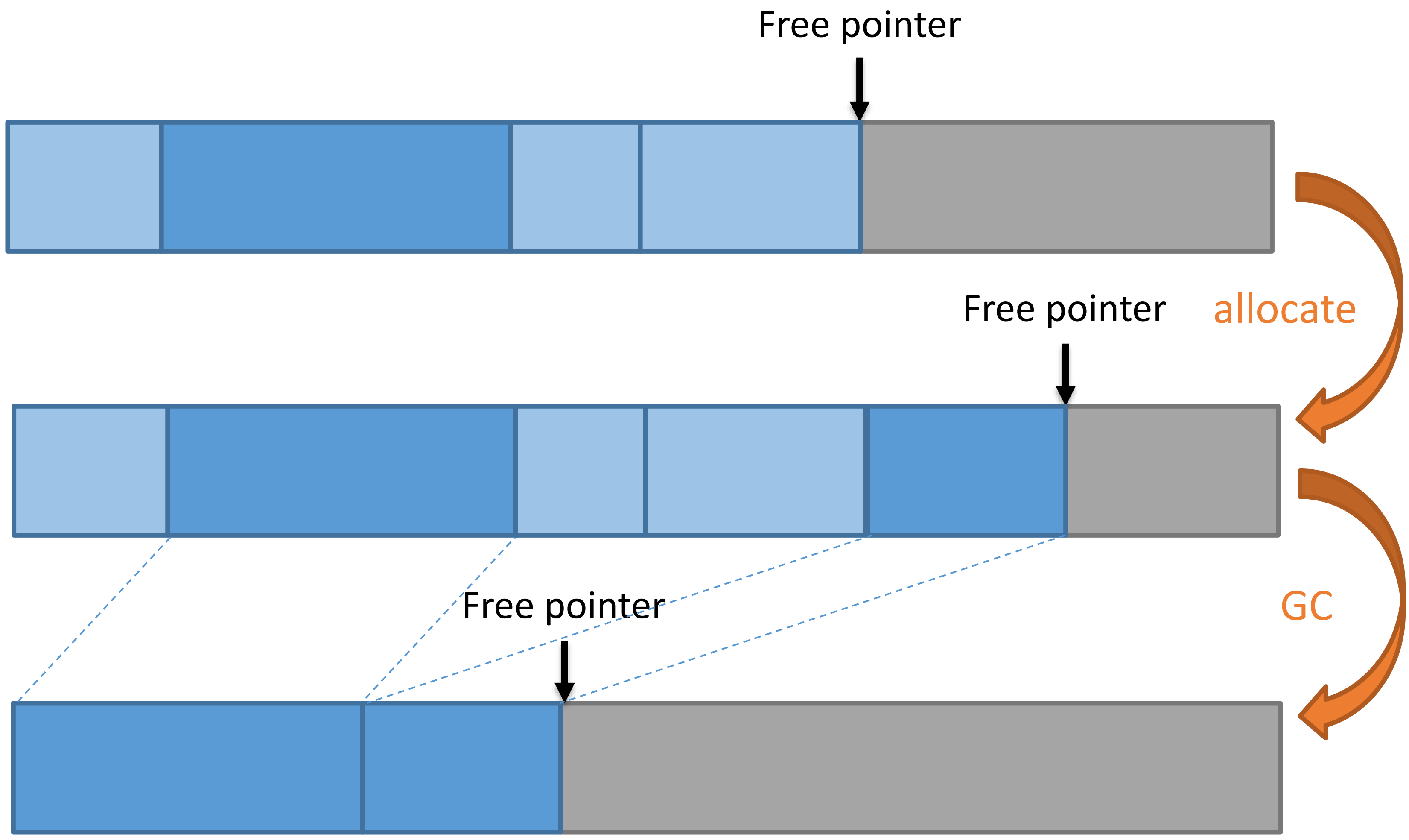
Other Possibilities

- Merge neighbor free blocks
  - Easily possible during sweep phase
- Buddy System
  - Discrete block sizes ordered by address
  - Exponential sizes (power of 2, Fibonacci)
  - Very fast merging & allocation & freeing
  - But huge internal fragmentation (unusable rests)
- Compacting Garbage Collection

Compacting GC

- Also called Mark & Copy GC
- Allocation at heap end (super-efficient)
- GC moves alive objects together
- Need to update all references on object moving

Compacting GC



Other Advanced GC Concepts

Finalizers

Incremental GC

Finalizer

- Method that runs before deletion of an object
  - Final cleanup: Close connections, dispose external resources etc.
- Initiated by GC when object is identified as garbage

```
class Block {
  @Override
  protected void finalize() {
     ...
  }
}
```

Java finalizer

Separate Finalization

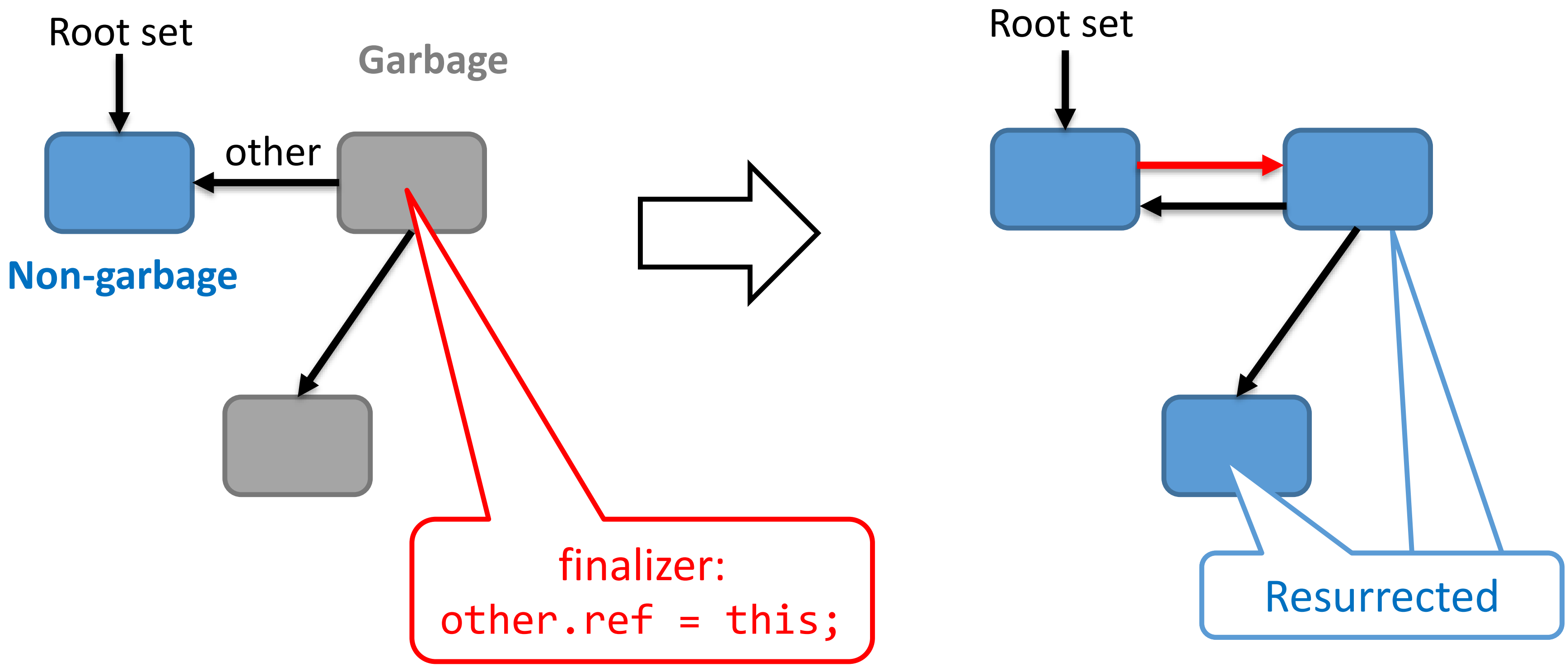
- Finalizer is not executed in GC phase, but only later
- Reasons:
  - Finalizer can take long time
    => blocks GC
  - Finalizer can allocate new object
    => corrupts GC
  - Programming bugs in finalizer
    => crashes GC
  - Finalizer can make garbage alive again
    => resurrection

Resurrection

- Finalizer can make an object alive again, after it has been garbage
- Not only own object but also indirectly other objects can resurrect

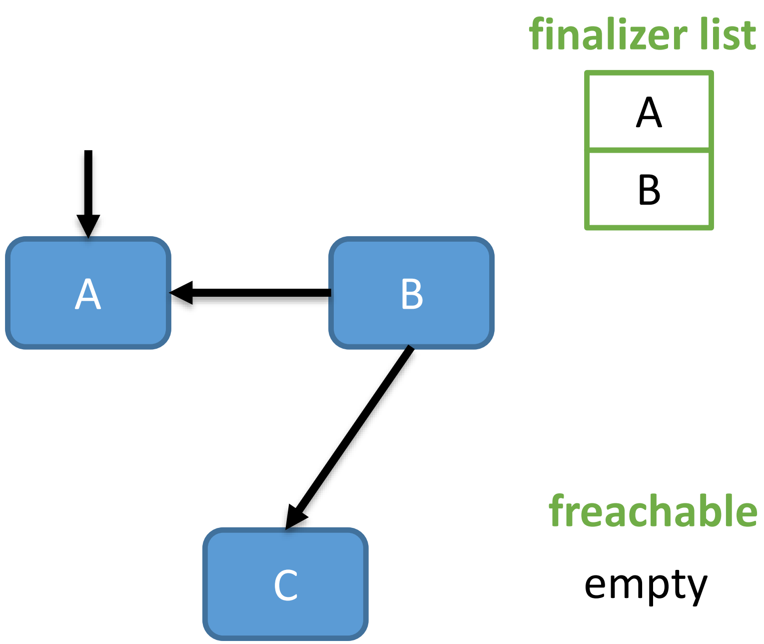
How is this possible?

Resurrection



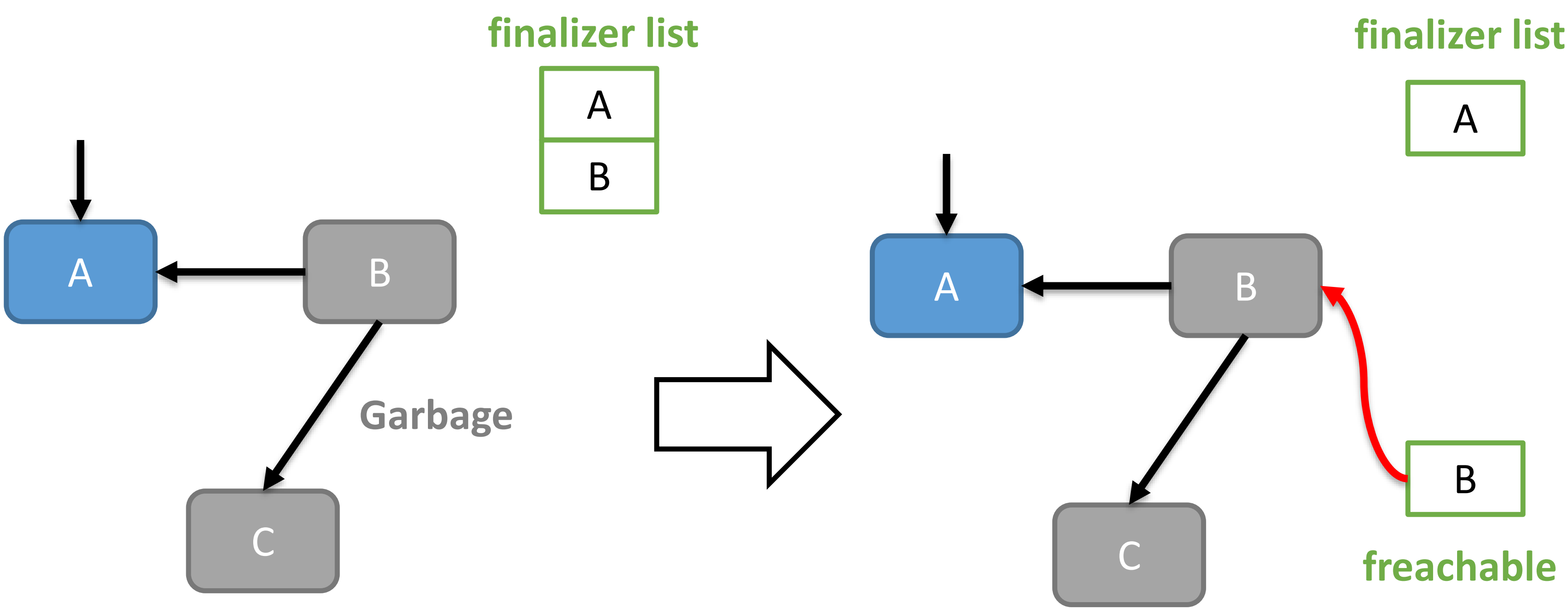
Finalizer Internals

- finalizer list = registered finalizers
- freachable list = pending finalizers to be executed



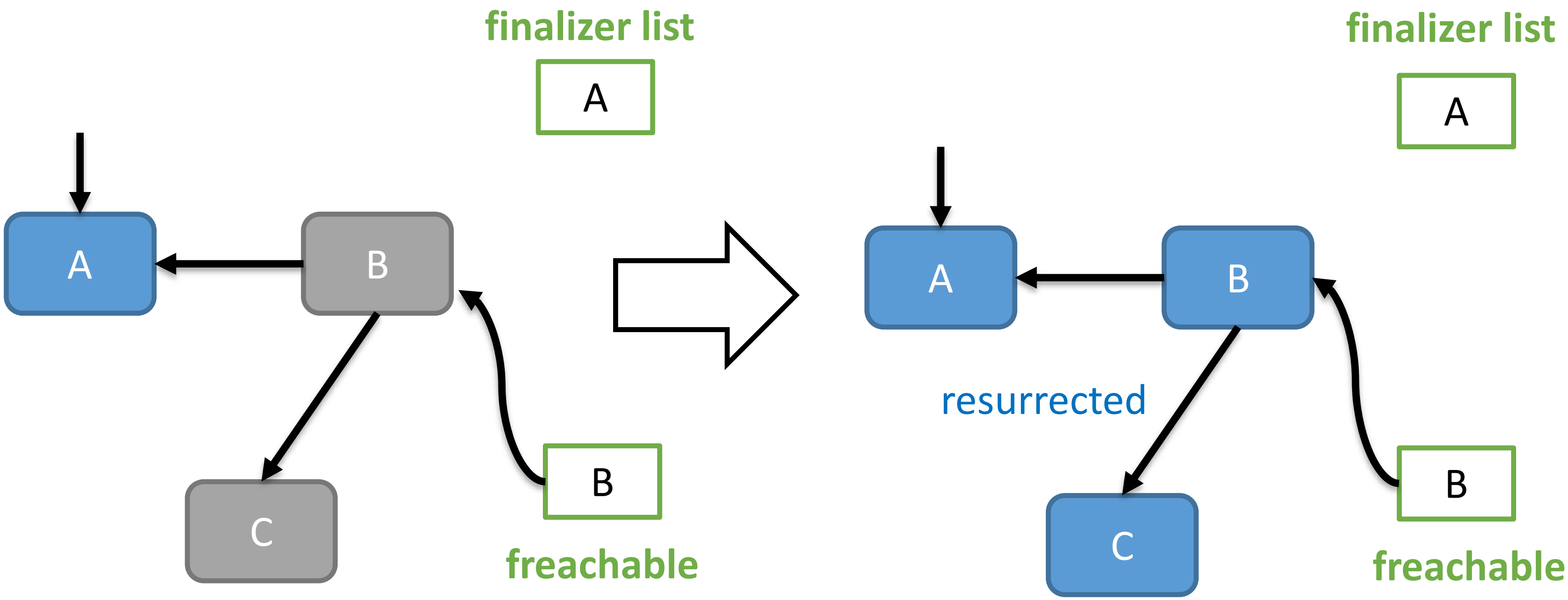
Finalizer Internals

- Garbage with finalizer is registered to freachable



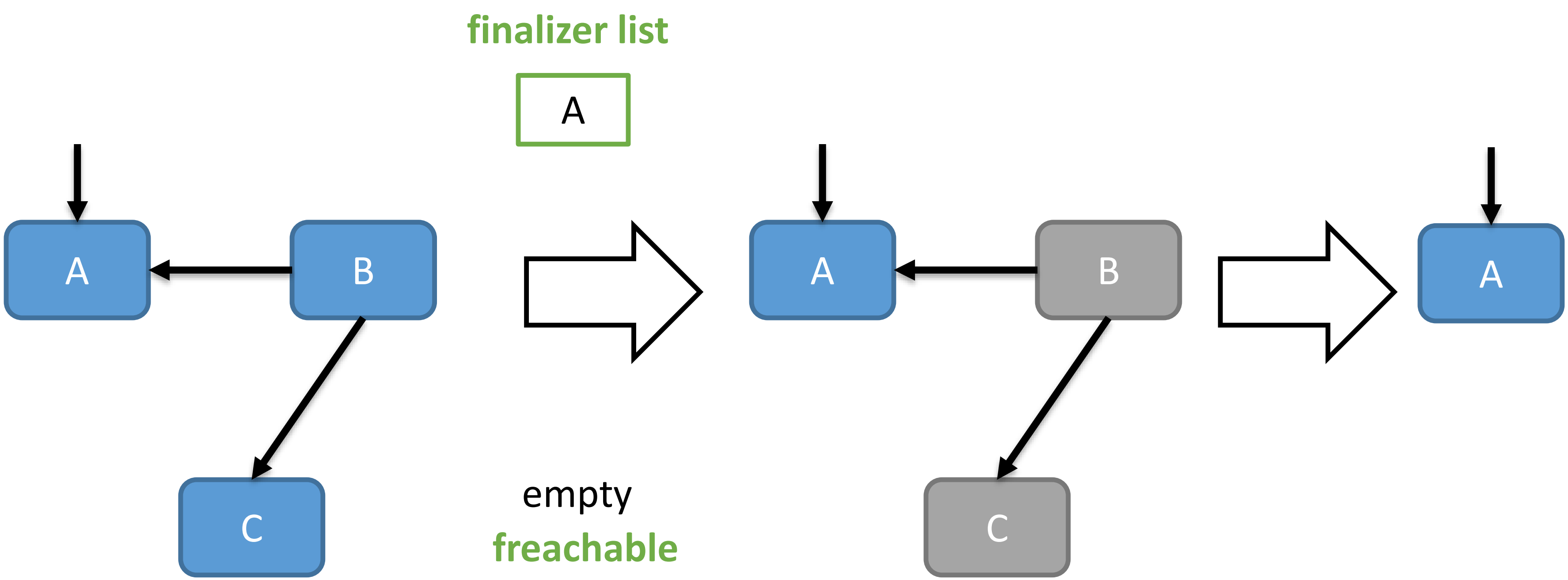
Finalizer Internals

- Insertion in freachable effects resurrection => new GC phase is necessary



Finalizer Internals

- Finalizer runs later => freachable entry is removed
- New GC run is necessary to finally free the object



Finalizer Impact

- GC needs 2 mark phases
  - Mark and detect garbage with finalizer
  - Mark again starting from freachable, then sweep
- Object with finalizer needs at least 2 GC runs until deletion
  - Free memory may not be reclaimed fast enough

```
System.gc();
System.runFinalization();
System.gc();
```

Finalizer Programming Aspects

- Order of finalizers is undefined
- Finalizer can run arbitrarily delayed
- Finalizer are concurrent to main program
  - Separate thread or arbitrary interleaving
- Does the finalizer run again after resurrection?
  - Not in Java

Incremental GC

- Stop & Go GC may cause too long interrupts
- Goal: Perform GC in smaller steps

Generational GC

Partitioned GC

… and many more …

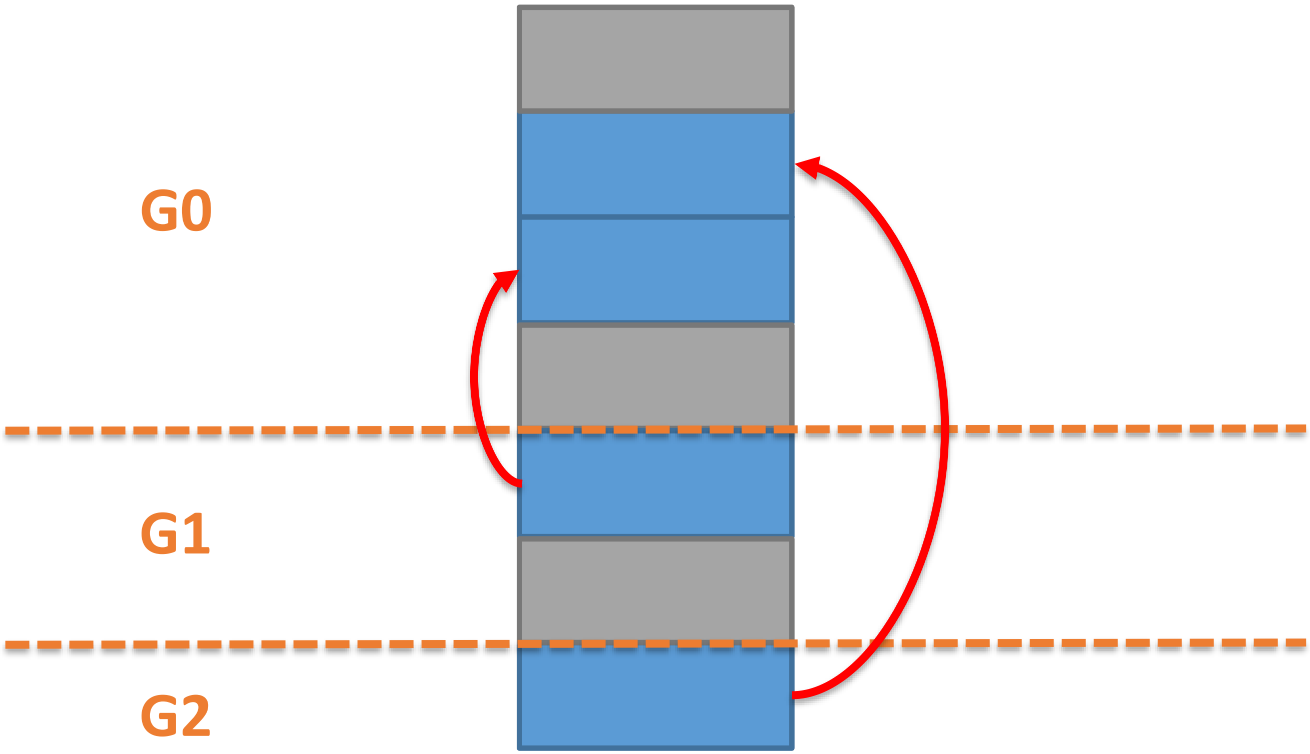
Generational GC

- Time mirror heuristics
  - Young objects ⇔ short expected lifetime
  - Old object ⇔ long expected lifetime

- 3 generations

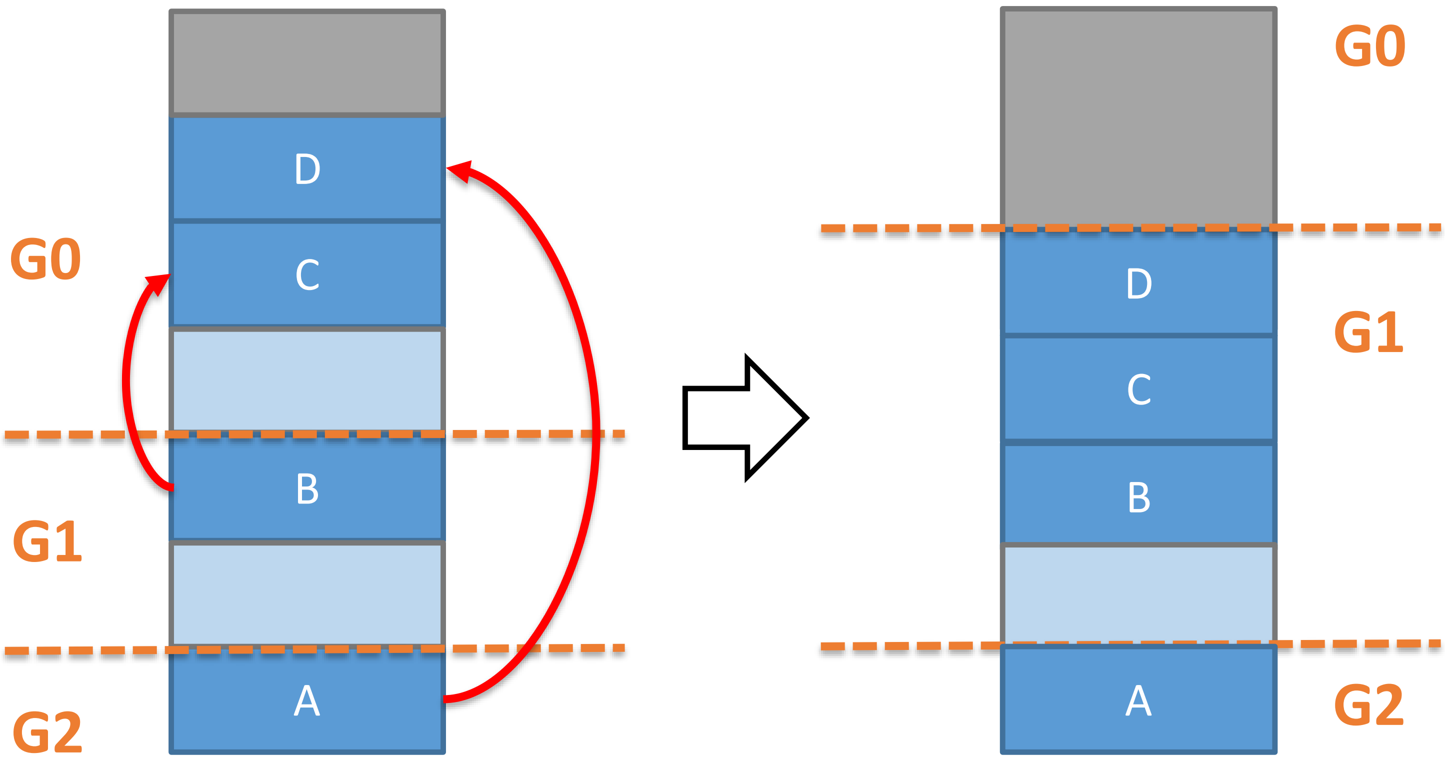
Age	Generation	GC frequency	GC pause
Young	G0	High	Short
Medium	G1	Medium	Medium
Old	G2	Low	Long

Heap with Generations



Additional root set for G0: All references pointing from G1 or G2 into G0

Collecting G0



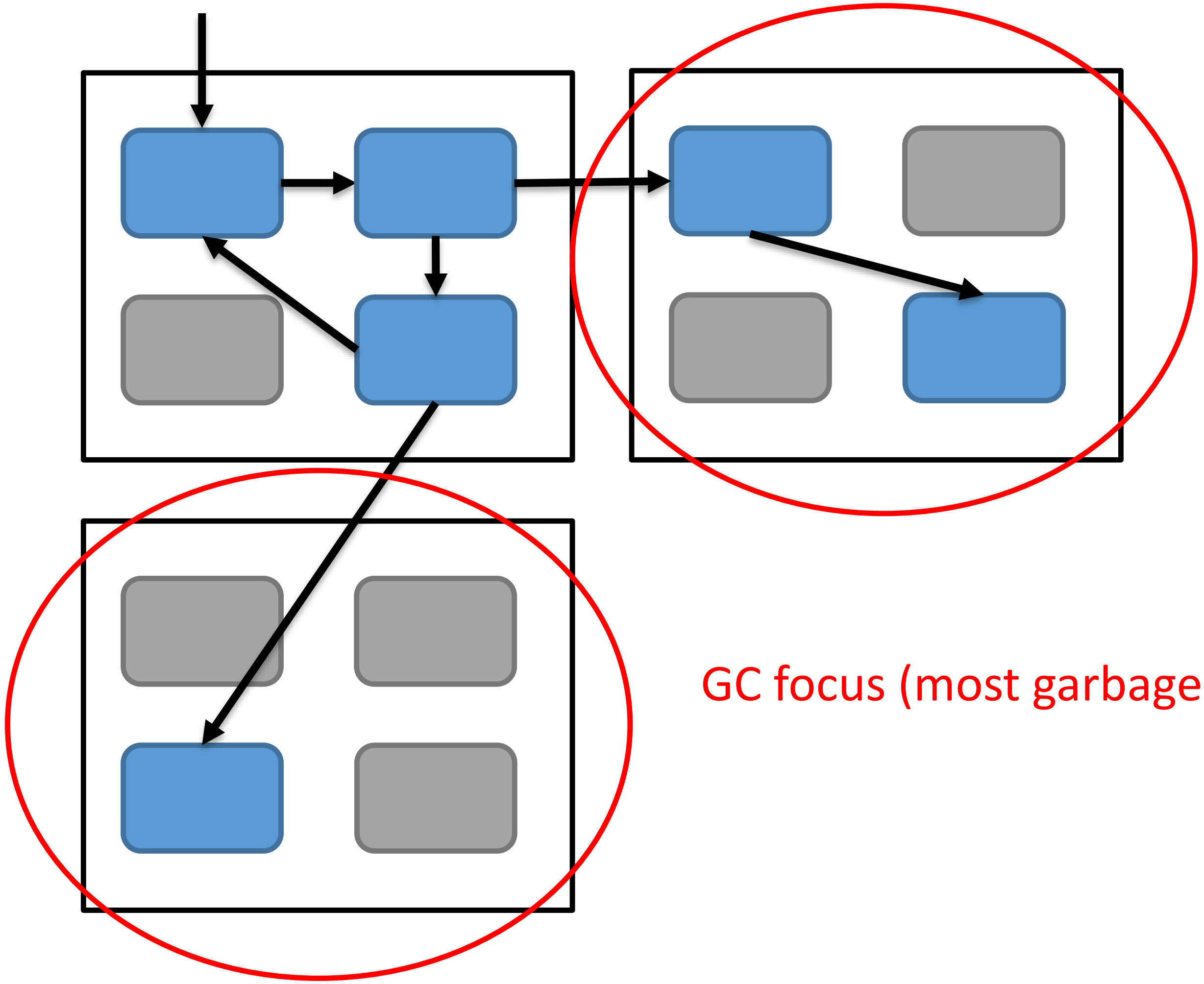
Root Sets with Generations

- References from old to new generations
  - Additional root set to new generations
- Write barriers: Detecting references writes in old generations
  - Software: Code instrumentation
  - Hardware: Read-only page protection => page fault
- GC on old generations must include new generations
  - G1 includes G0, G2 collection involves entire heap

Java: G1 Partitioned GC

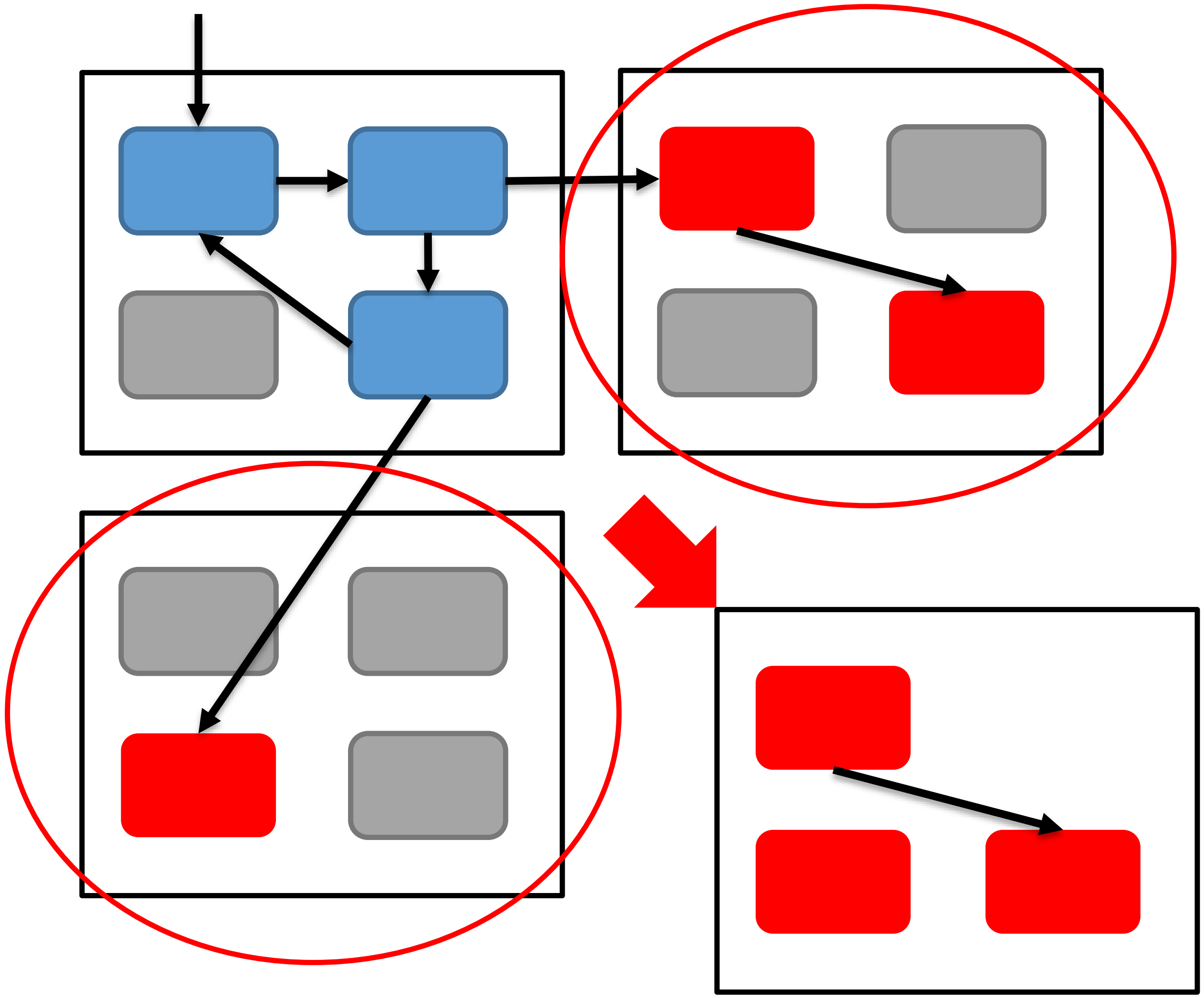
- Organize heap in partitions
  – Goal: Short GC interruptions
- Concurrent marking with snapshots
  – Detect relevant concurrent updates
- Focus GC on partitions with most inner garbage ("garbage first")
  – Evacuate alive objects in new partition
- Problem: cyclic garbage across partitions
  – Still requires full GC ("stop the world")

Partitioned GC

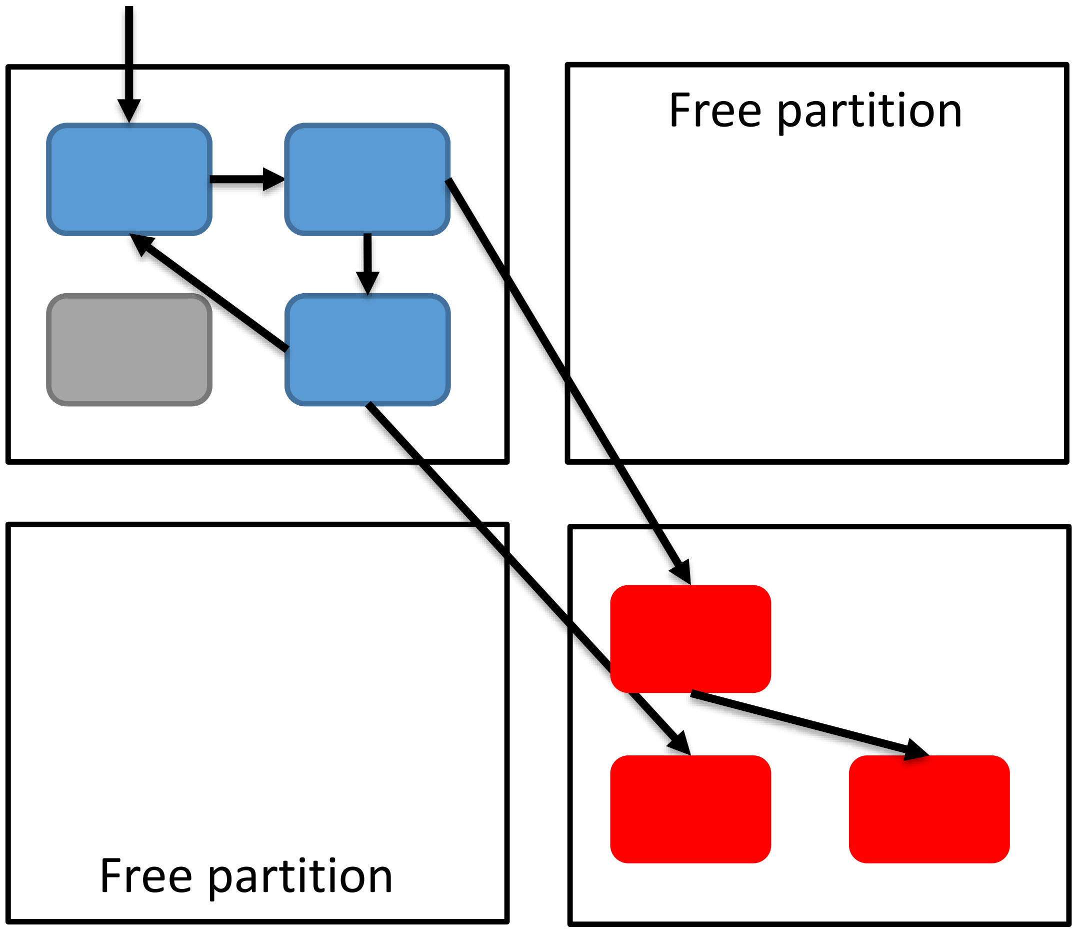


GC focus (most garbage)

Partitioned GC



Partitioned GC



Review: Learning Goals

- ✓ Understand how free heap blocks are managed
- ✓ Gain principal knowledge of advanced GC mechanisms

Further Reading

- Dragon Book, Garbage Collection
  - Section 7.6.4-7.6.5: Mark and compact, mark and copy
  - Section 7.7.4: Generational GC
- Optional, if interested
  - R. Jones, A. Hosking und E. Moss. The Garbage Collection Handbook. Chapman & Hall, 2011
  - Java G1 (Garbage First) GC
    - http://www.oracle.com/technetwork/tutorials/tutorials-1876574.html
  - Jeffrey Richter. Garbage Collection: Automatic Memory Management in .NET, MSDN Magazine, Nov. & Dec. 2000
    - Finalizer, Weak References, Compacting GC