

# Garbage Collection

Reference Counting

Mark-and-Sweep

Short-Pause Methods

# The Essence

- ◆ Programming is easier if the run-time system “**garbage-collects**” --- makes space belonging to unusable data available for reuse.
  - ◆ Java does it; C does not.
  - ◆ But stack allocation in C gets some of the advantage.

# Desiderata

1. Speed --- low overhead for garbage collector.
2. Little program interruption.
  - ◆ Many collectors shut down the program to hunt for garbage.
3. *Locality* --- data that is used together is placed together on pages, cache-lines.

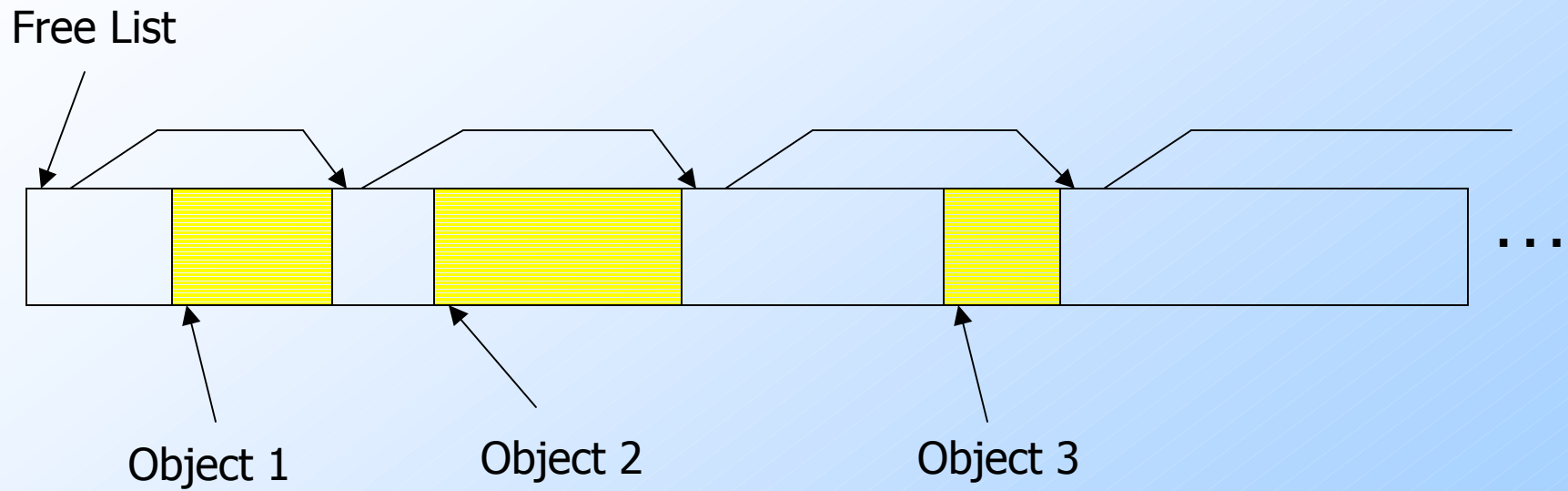
# The Model --- (1)

- ◆ There is a *root set* of data that is a-priori reachable.
  - ◆ **Example**: In Java, root set = static class variables plus variables on run-time stack.
- ◆ *Reachable data* : root set plus anything referenced by something reachable.
- ◆ **Question**: Why doesn't this make sense for C? Why is it OK for Java?

# The Model --- (2)

- ◆ Things requiring space are “objects.”
- ◆ Available space is in a *heap* --- large area managed by the run-time system.
  - ◆ *Allocator* finds space for new objects.
    - Space for an object is a *chunk*.
  - ◆ *Garbage collector* finds unusable objects, returns their space to the heap, and maybe moves objects around in the heap.

# A Heap



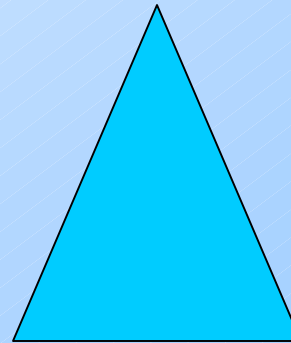
# Taxonomy

## Garbage Collectors

Reference-  
Counters



Trace-  
Based



# Reference Counting

- ◆ The simplest (but imperfect) method is to give each object a *reference count* = number of references to this object.
  - ◆ OK if objects have no internal references.
- ◆ Initially, object has one reference.
- ◆ If reference count becomes 0, object is garbage and its space becomes available.



# Examples

```
Integer i = new Integer(10);
```

- ◆ Integer object is created with RC = 1.

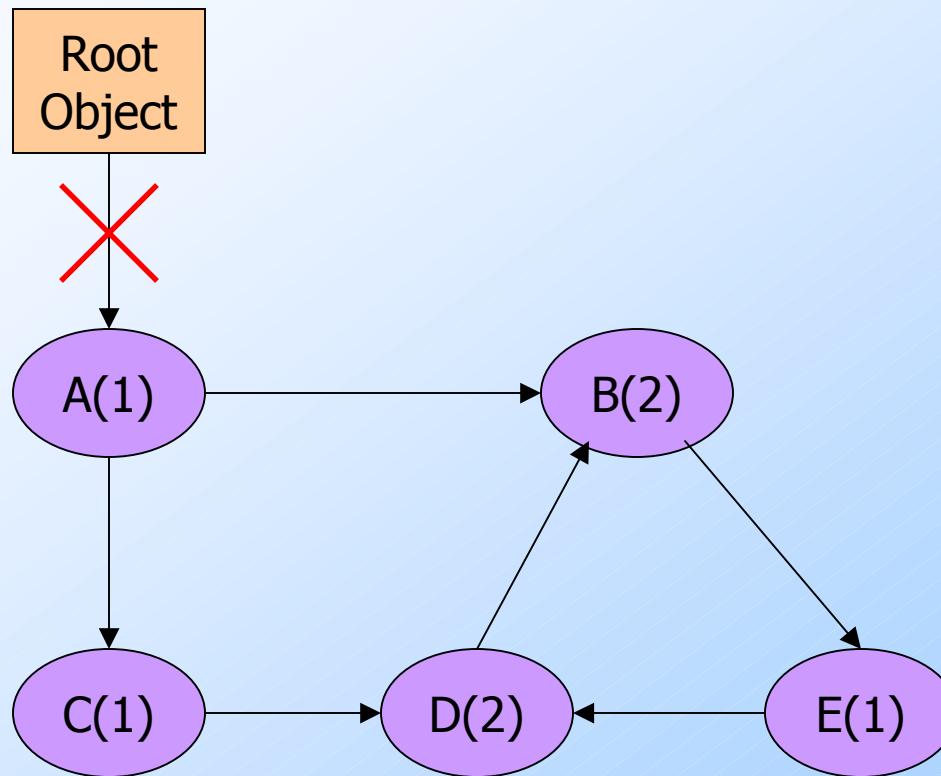
```
j = k; (j, k are Integer references.)
```

- ◆ Object referenced by j has RC--.
- ◆ Object referenced by k has RC++.

# Transitive Effects

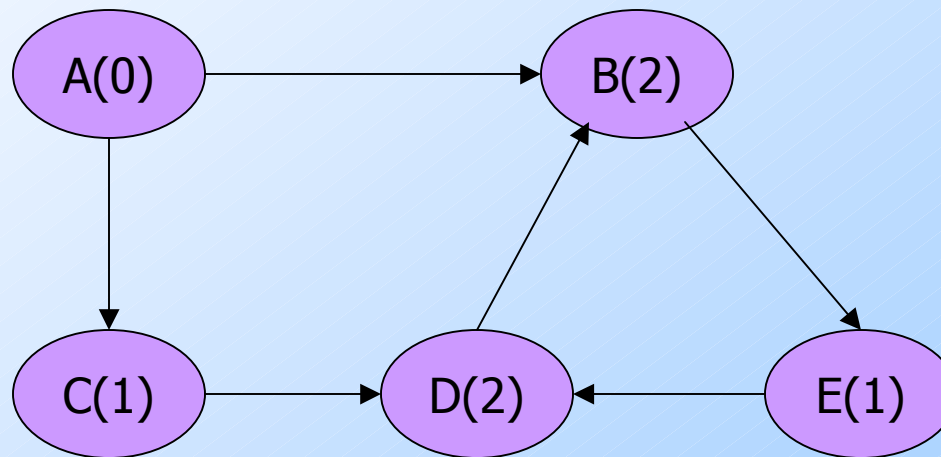
- ◆ If an object reaches  $RC=0$  and is collected, the references within that object disappear.
- ◆ Follow these references and decrement  $RC$  in the objects reached.
- ◆ That may result in more objects with  $RC=0$ , leading to recursive collection.

# Example: Reference Counting



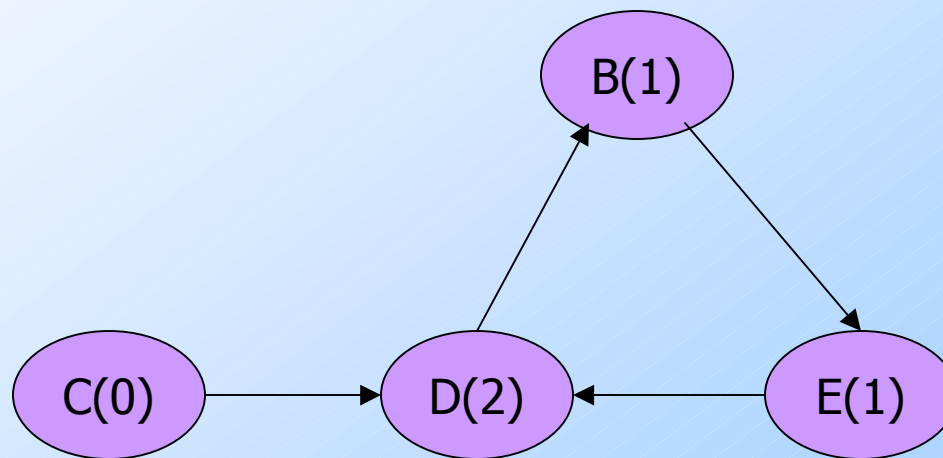
# Example: Reference Counting

Root  
Object



# Example: Reference Counting

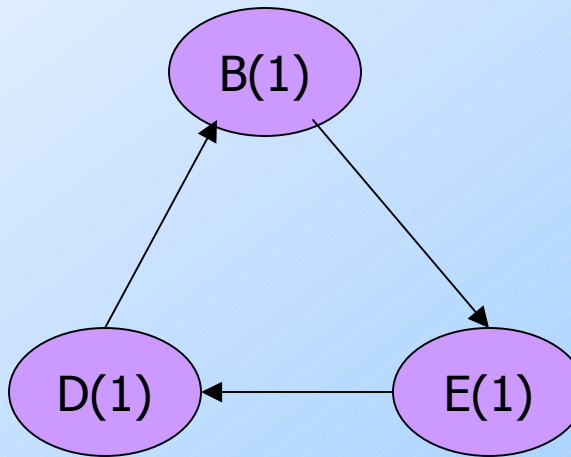
Root  
Object



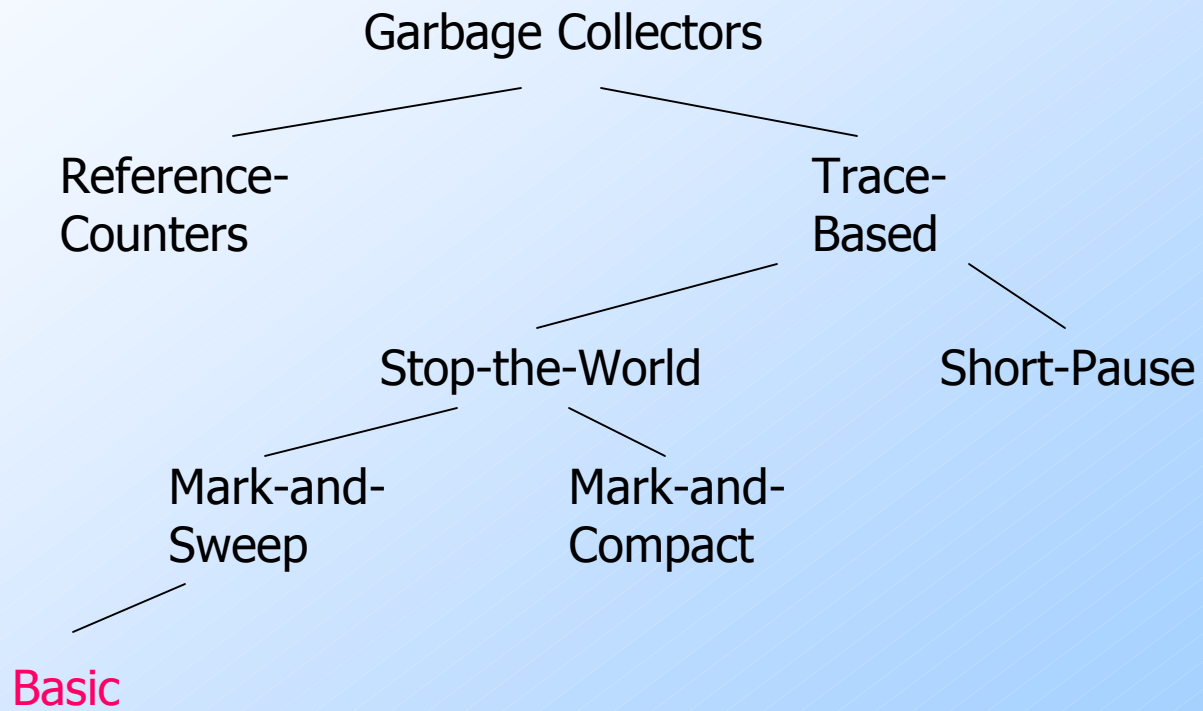
# Example: Reference Counting

Root  
Object

B, D, and E are garbage, but their reference counts are all  $> 0$ . They never get collected.



# Taxonomy



# Four States of Memory Chunks

1. *Free* = not holding an object; available for allocation.
2. *Unreached* = Holds an object, but has not yet been reached from the root set.
3. *Unscanned* = Reached from the root set, but its references not yet followed.
4. *Scanned* = Reached and references followed.



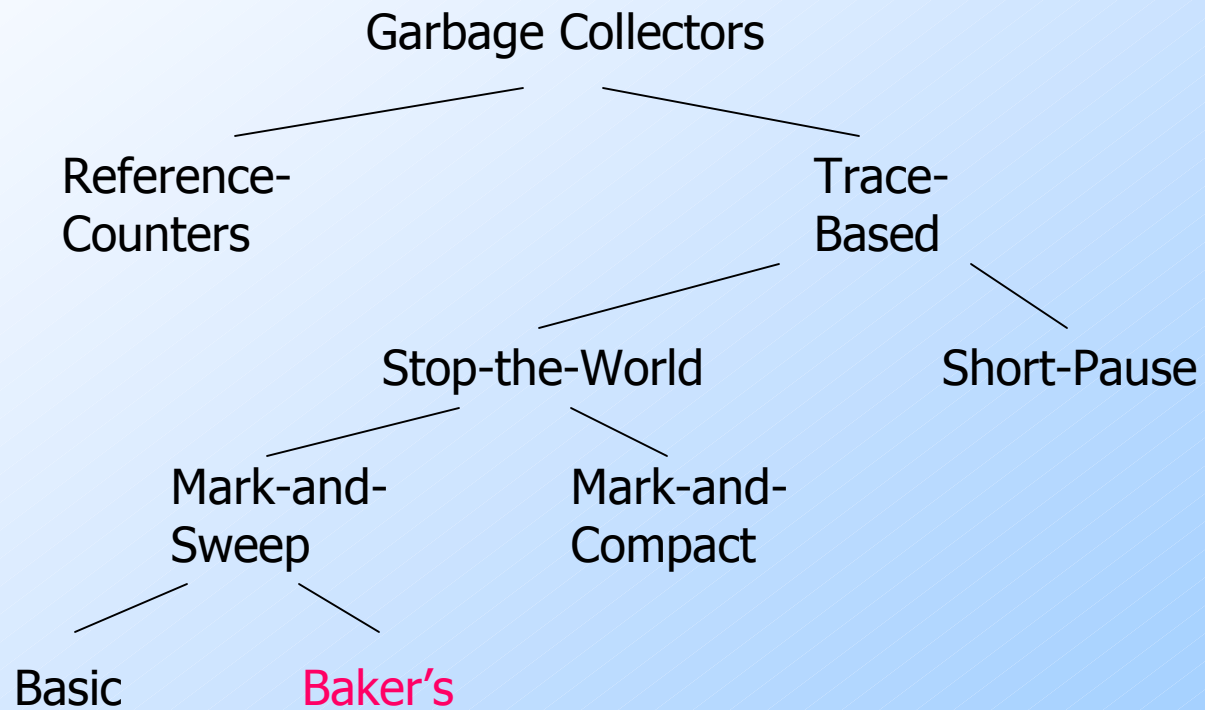
# Marking

1. Assume all objects in **Unreached** state.
2. Start with the root set. Put them in state **Unscanned**.
3. **while** **Unscanned** objects remain **do**  
    examine one of these objects;  
    make its state be **Scanned**;  
    add all referenced objects to **Unscanned**  
    if they have not been there;  
**end;**

# Sweeping

- ◆ Place all objects still in the **Unreached** state into the **Free** state.
- ◆ Place all objects in **Scanned** state into the **Unreached** state.
  - ◆ To prepare for the next mark-and-sweep.

# Taxonomy



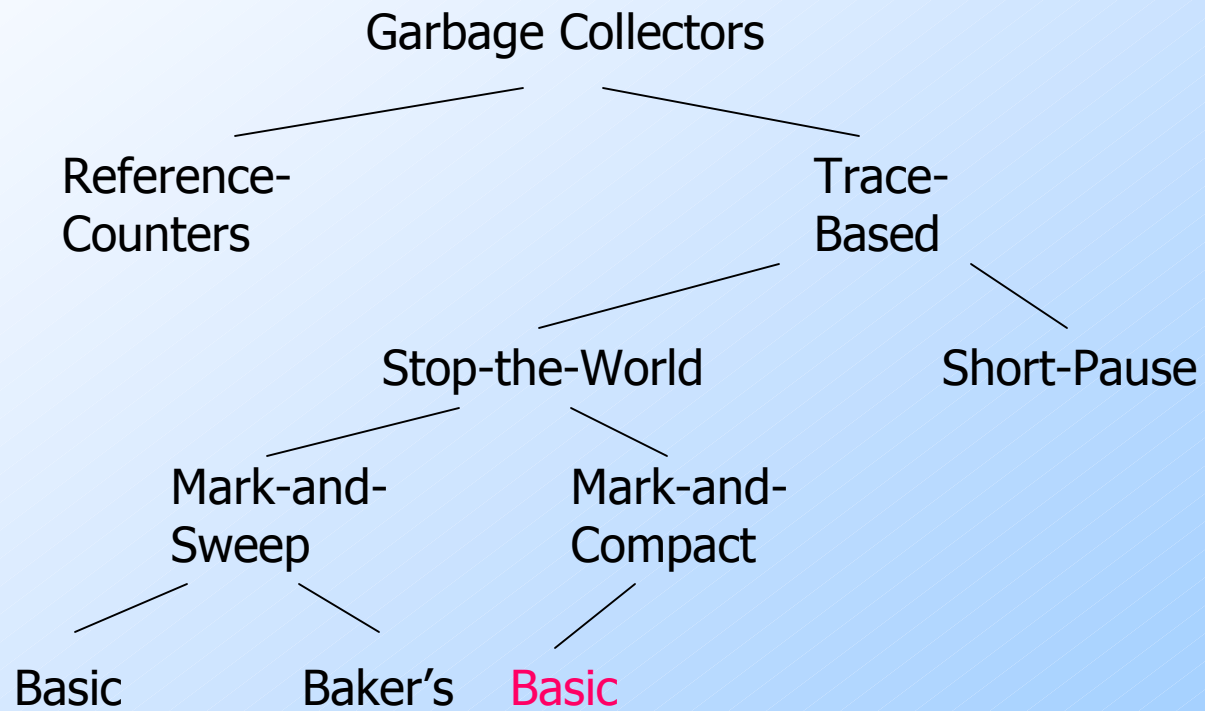
# Baker's Algorithm --- (1)

- ◆ **Problem:** The basic algorithm takes time proportional to the heap size.
  - ◆ Because you must visit all objects to see if they are **Unreached**.
- ◆ Baker's algorithm keeps a list of all allocated chunks of memory, as well as the **Free** list.

# Baker's Algorithm --- (2)

- ◆ **Key change:** In the sweep, look only at the list of allocated chunks.
- ◆ Those that are not marked as **Scanned** are garbage and are moved to the **Free** list.
- ◆ Those in the **Scanned** state are put in the **Unreached** state.
  - ◆ For the next collection.

# Taxonomy



# Issue: Why Compact?

- ◆ *Compact* = move reachable objects to contiguous memory.
- ◆ *Locality* --- fewer pages or cache-lines needed to hold the active data.
- ◆ *Fragmentation* --- available space must be managed so there is space to store large objects.

# Mark-and-Compact

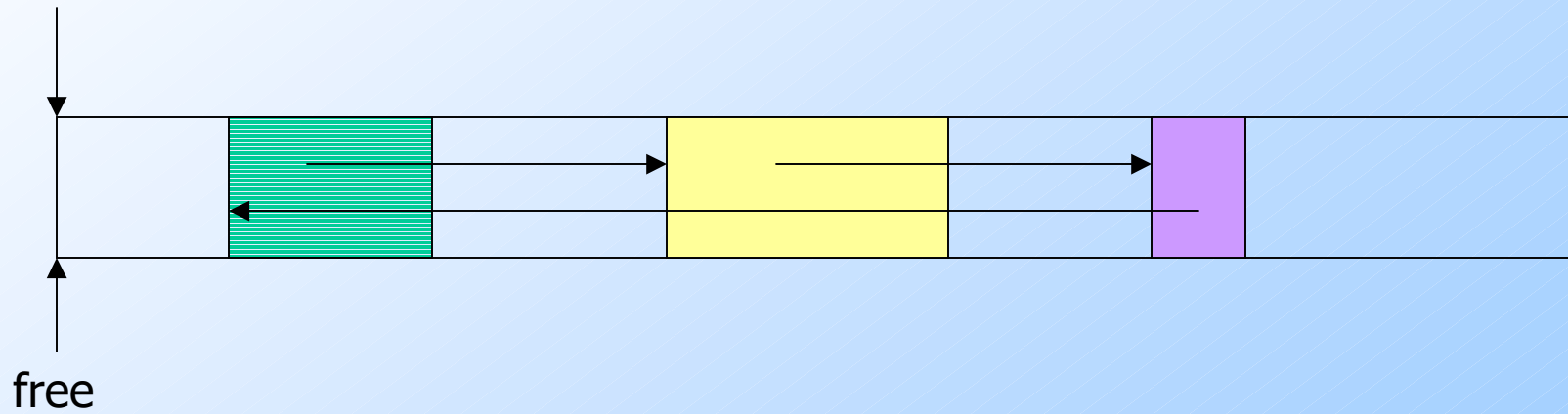
1. Mark reachable objects as before.
2. Maintain a table (hash?) from reached chunks to new locations for the objects in those chunks.
  - ◆ Scan chunks from low end of heap.
  - ◆ Maintain pointer *free* that counts how much space is used by reached objects so far.



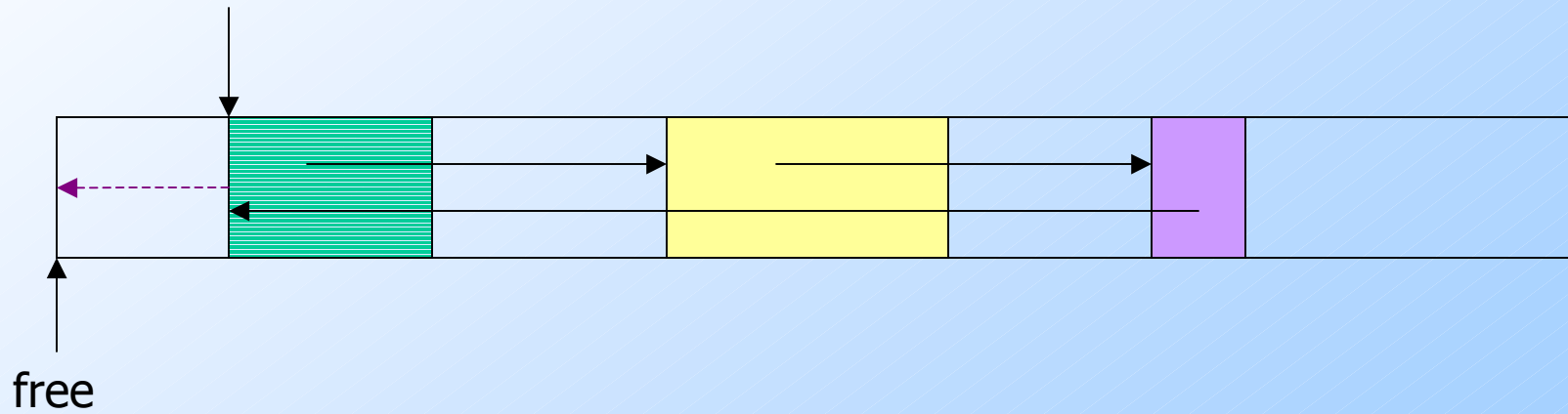
# Mark-and-Compact --- (2)

3. Move all reached objects to their new locations, and also retarget all references in those objects to the new locations.
  - ◆ Use the table of new locations.
4. Retarget root references.

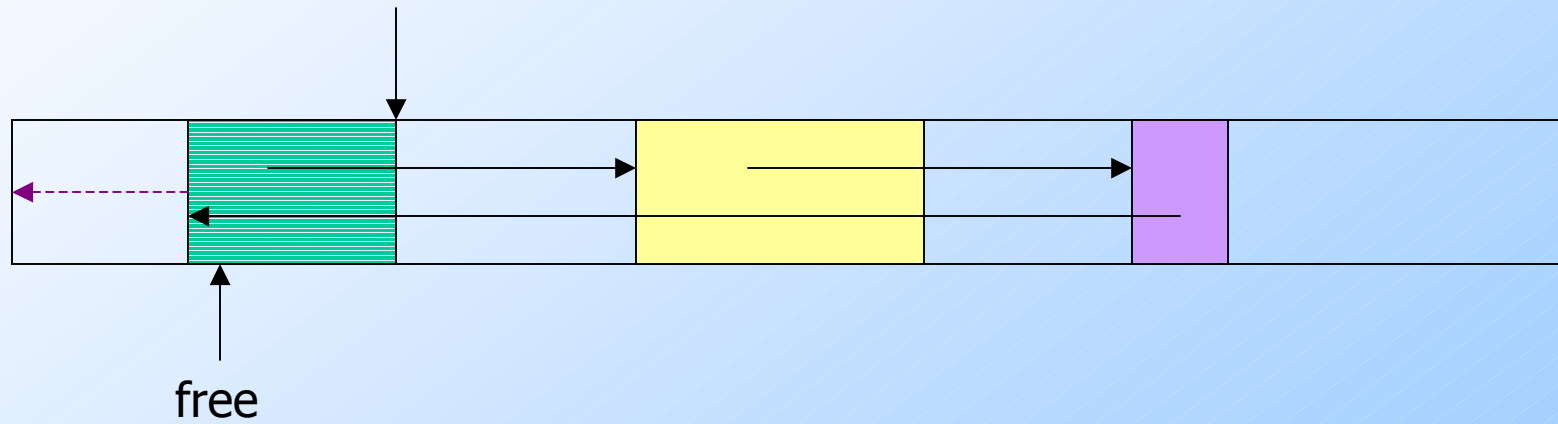
# Example: Mark-and-Compact



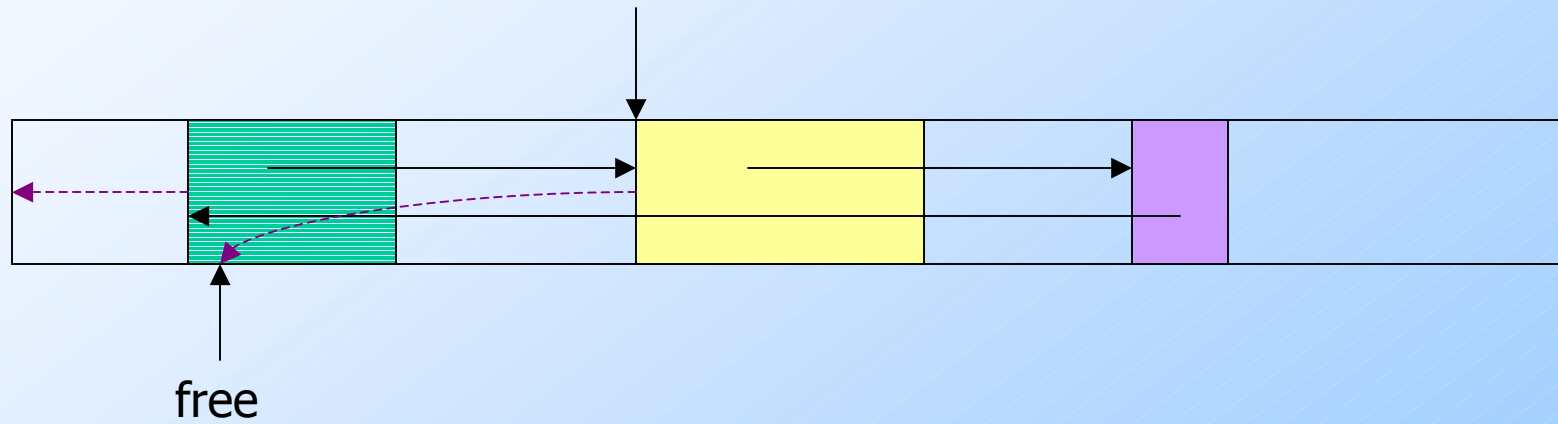
# Example: Mark-and-Compact



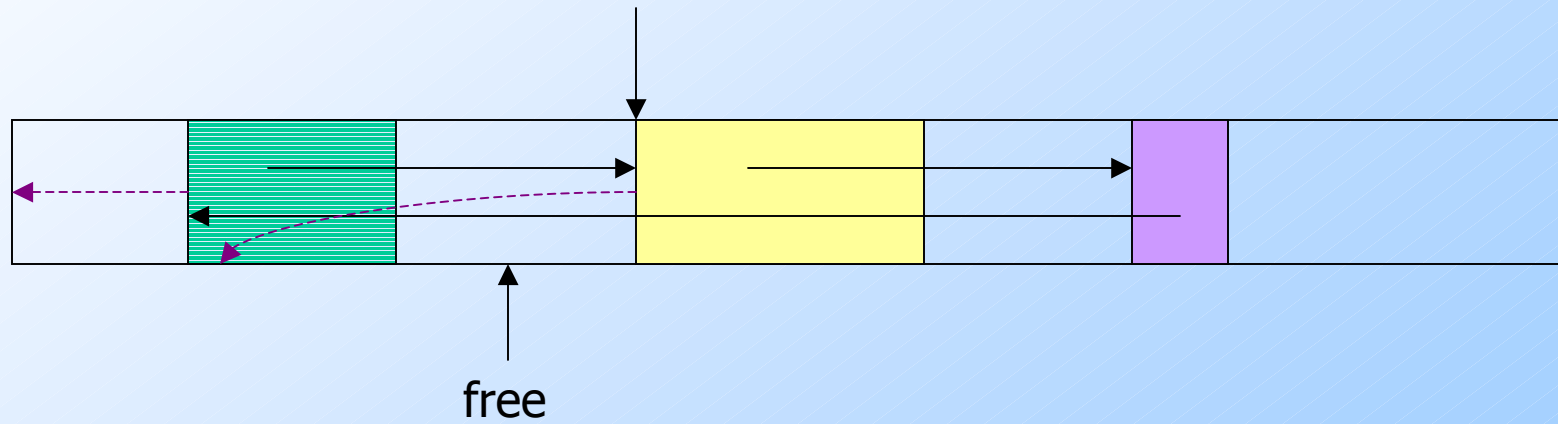
# Example: Mark-and-Compact



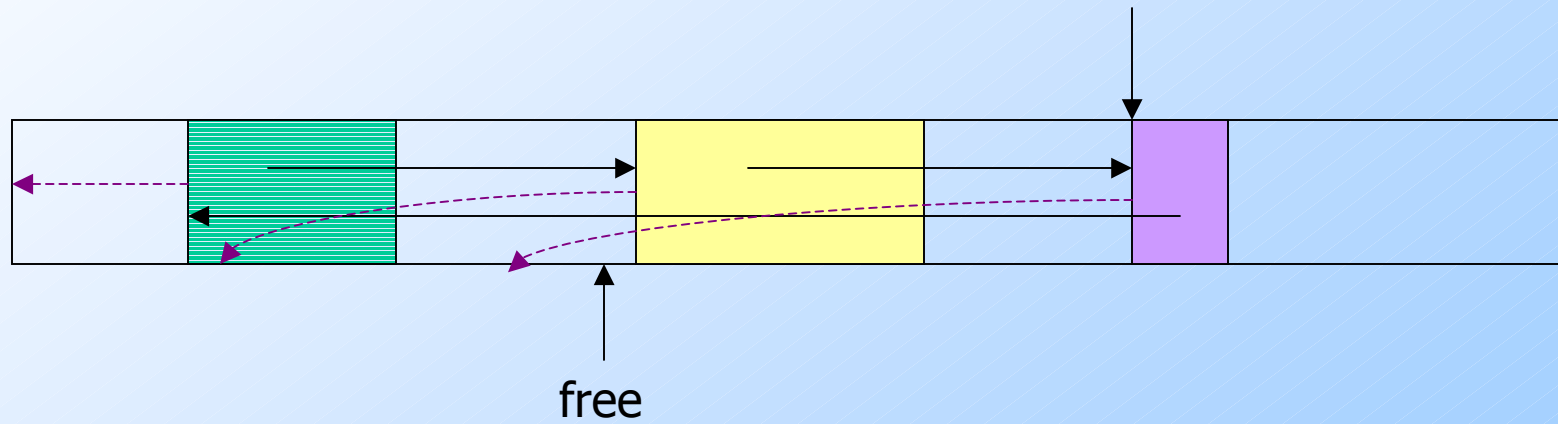
# Example: Mark-and-Compact



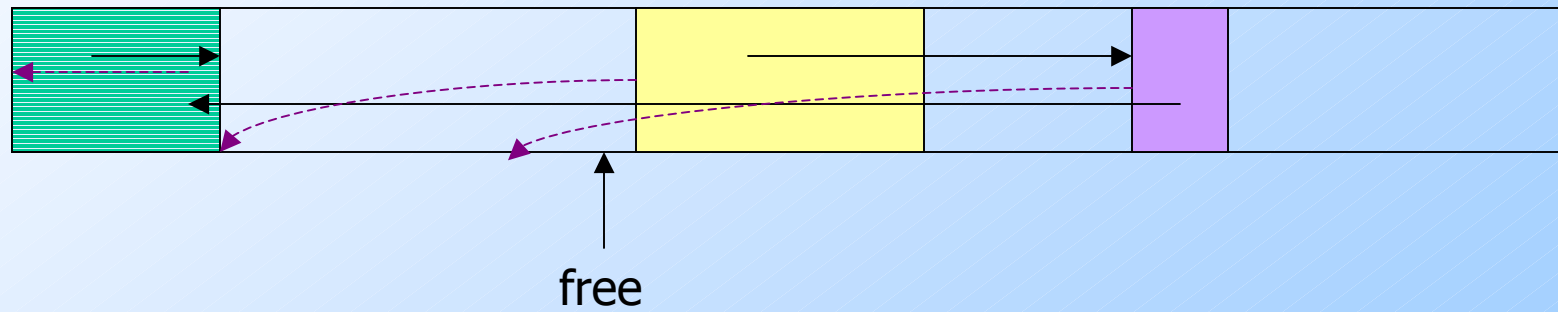
# Example: Mark-and-Compact



# Example: Mark-and-Compact

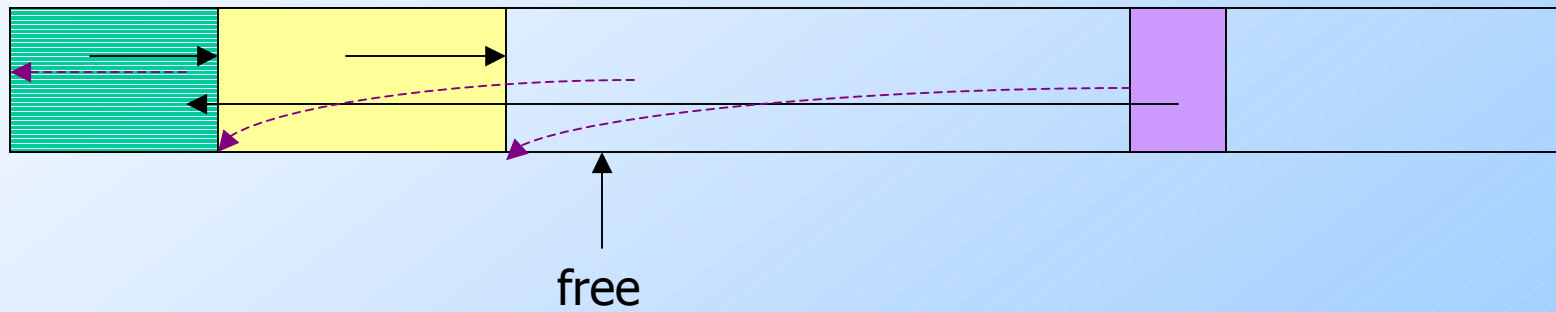


# Example: Mark-and-Compact

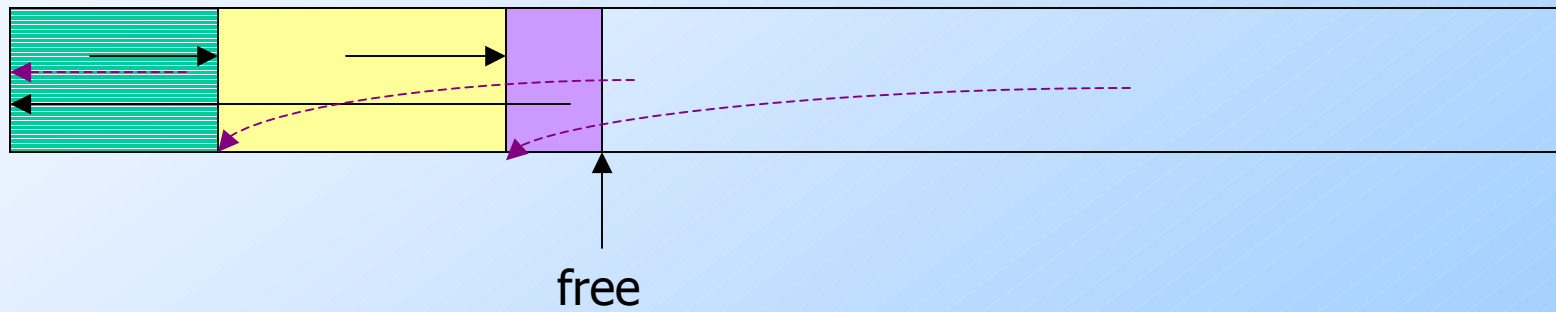




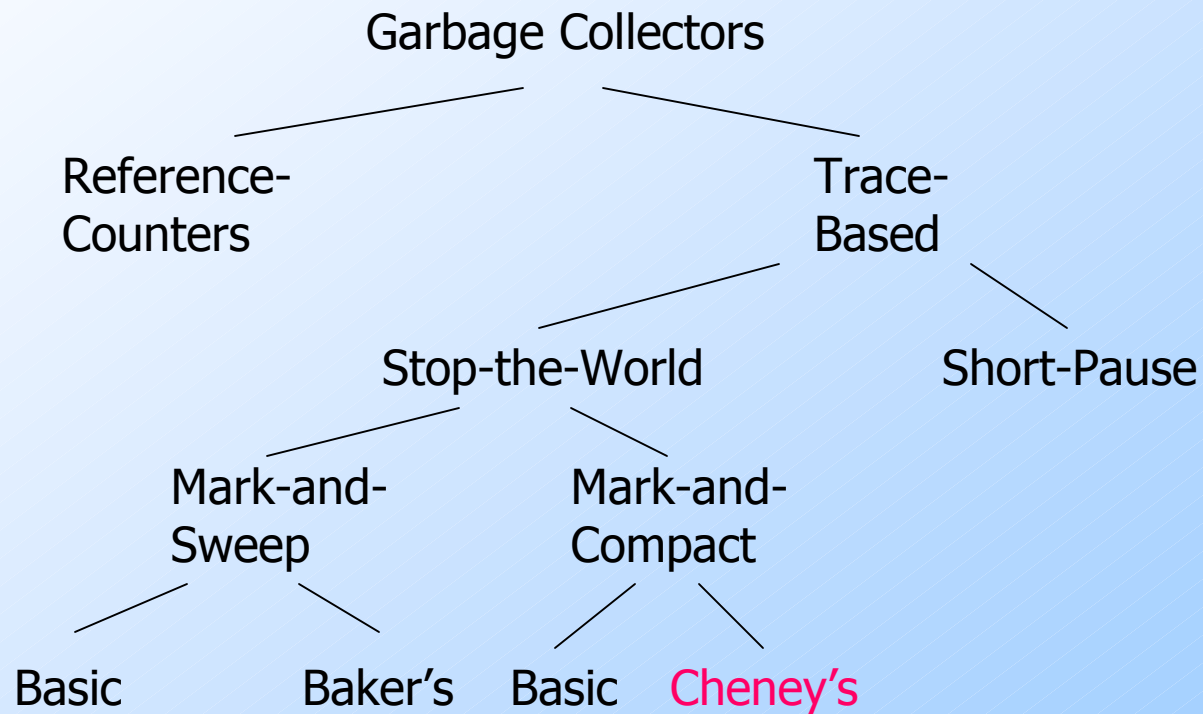
# Example: Mark-and-Compact



# Example: Mark-and-Compact



# Taxonomy

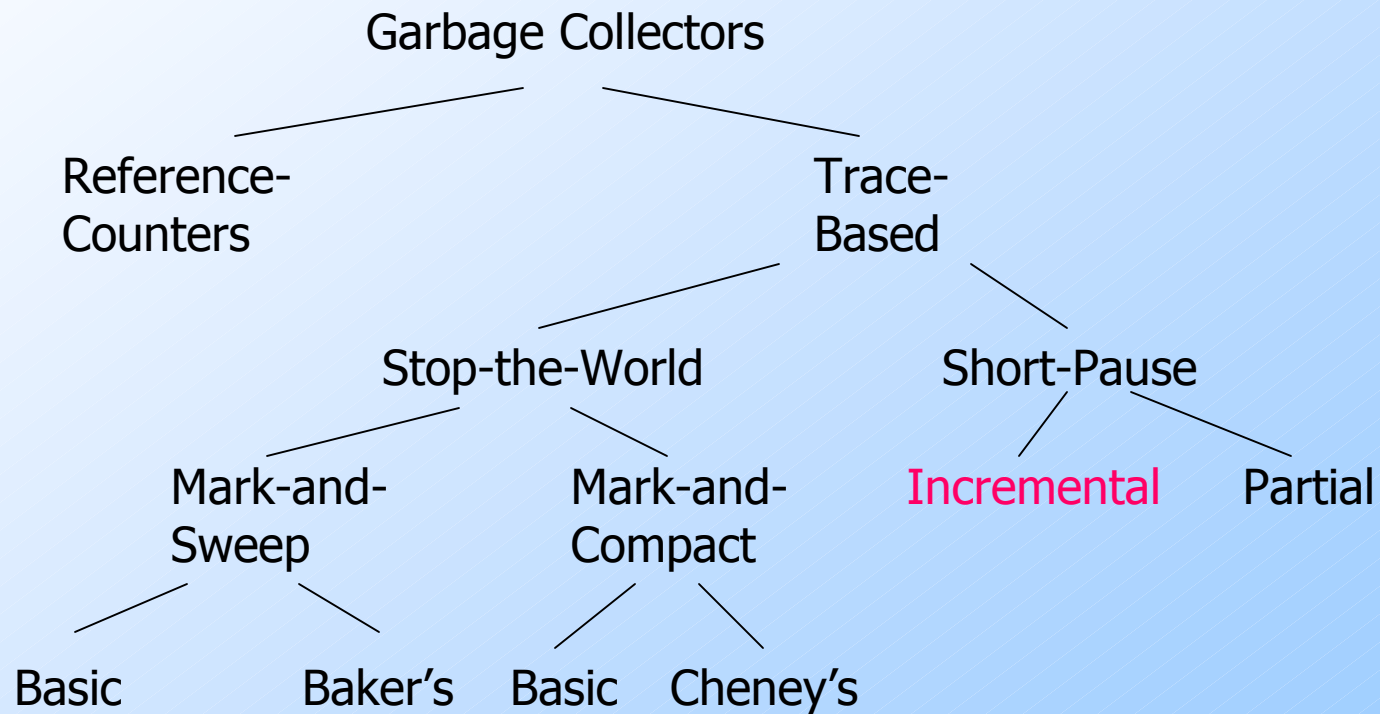


A different Cheney, BTW, so no jokes, please.

# Cheney's Copying Collector

- ◆ A shotgun approach to GC.
- ◆ 2 heaps: Allocate space in one, copy to second when first is full, then swap roles.
- ◆ Maintain table of new locations.
- ◆ As soon as an object is reached, give it the next free chunk in the second heap.
- ◆ As you scan objects, adjust their references to point to second heap.

# Taxonomy



# Short-Pause Garbage-Collection

1. *Incremental* --- run garbage collection in parallel with *mutation* (operation of the program).
2. *Partial* --- stop the mutation, but only briefly, to garbage collect a **part** of the heap.

# Problem With Incremental GC

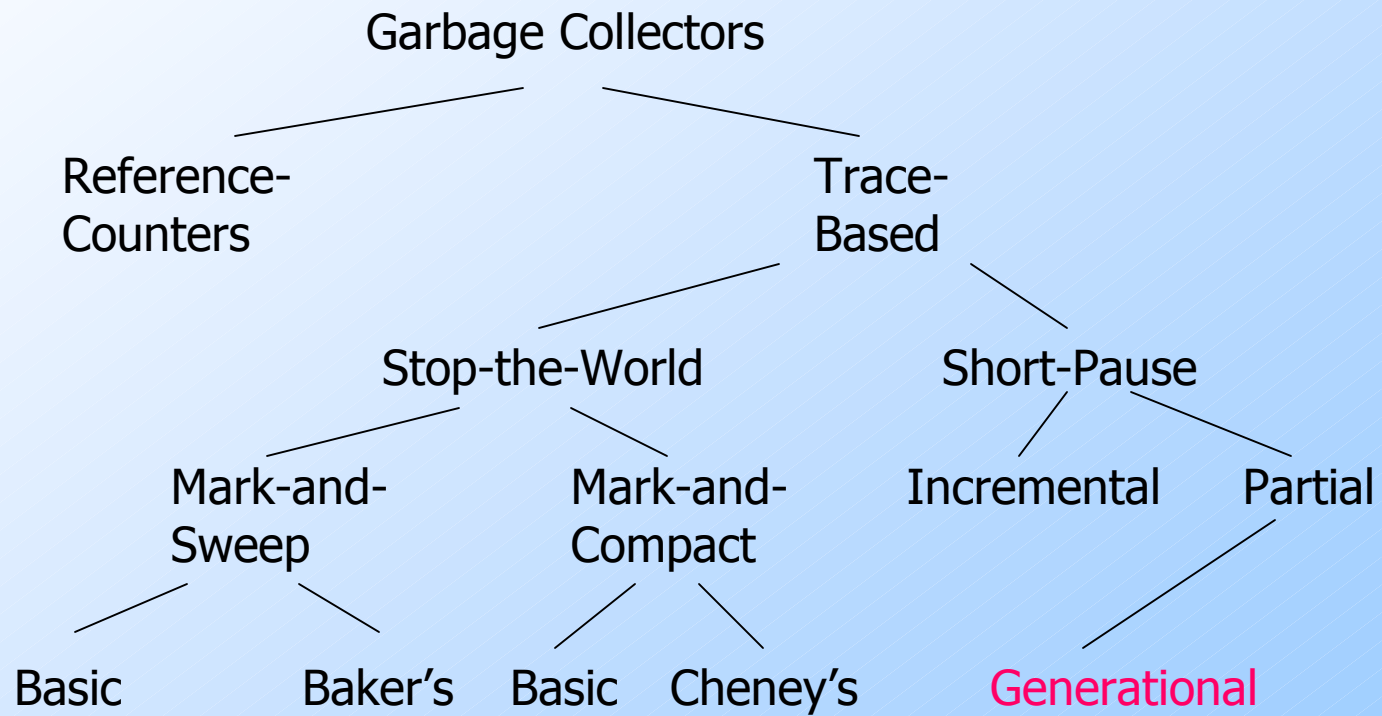
- ◆ OK to mark garbage as reachable.
- ◆ Not OK to GC a reachable object.
- ◆ If a reference **r** within a **Scanned** object is mutated to point to an **Unreached** object, the latter may be garbage-collected anyway.
  - ◆ **Subtle point**: How do you point to an **Unreached** object?

# One Solution: *Write Barriers*

- ◆ Intercept every write of a reference in a scanned object.
- ◆ Place the new object referred to on the **Unscanned** list.
- ◆ **A trick**: protect all pages containing **Scanned** objects.
  - ◆ A hardware interrupt will invoke the fixup.



# Taxonomy



# The Object Life-Cycle

- ◆ “Most objects die young.”
  - ◆ But those that survive one GC are likely to survive many.
- ◆ Tailor GC to spend more time on regions of the heap where objects have just been created.
  - ◆ Gives a better ratio of reclaimed space per unit time.

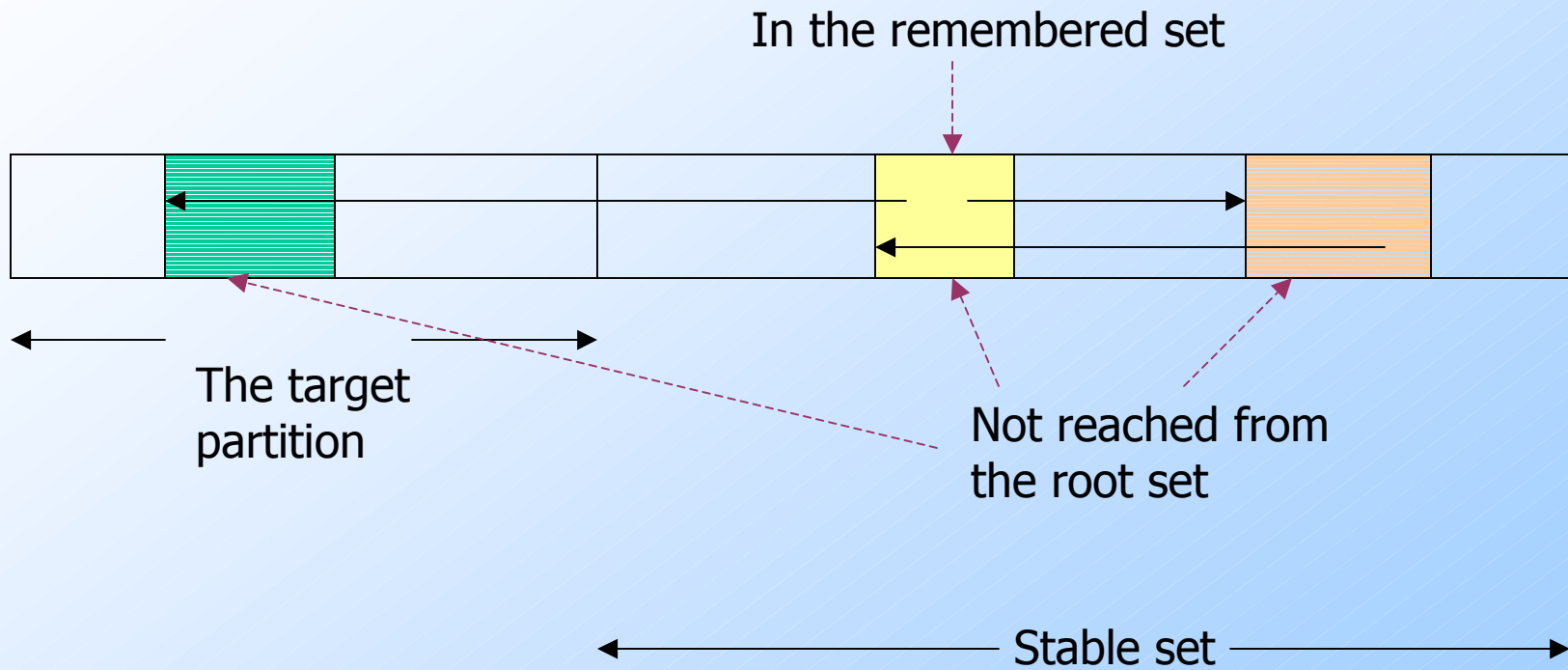
# Partial Garbage Collection

- ◆ We collect one part(ition) of the heap.
  - ◆ The *target* set.
- ◆ We maintain for each partition a *remembered* set of those objects outside the partition (the *stable* set) that refer to objects in the target set.
  - ◆ Write barriers can be used to maintain the remembered set.

# Collecting a Partition

- ◆ To collect a part of the heap:
  1. Add the remembered set for that partition to the root set.
  2. Do a reachability analysis as before.
- ◆ Note the resulting **Scanned** set may include garbage.

# Example: "Reachable" Garbage



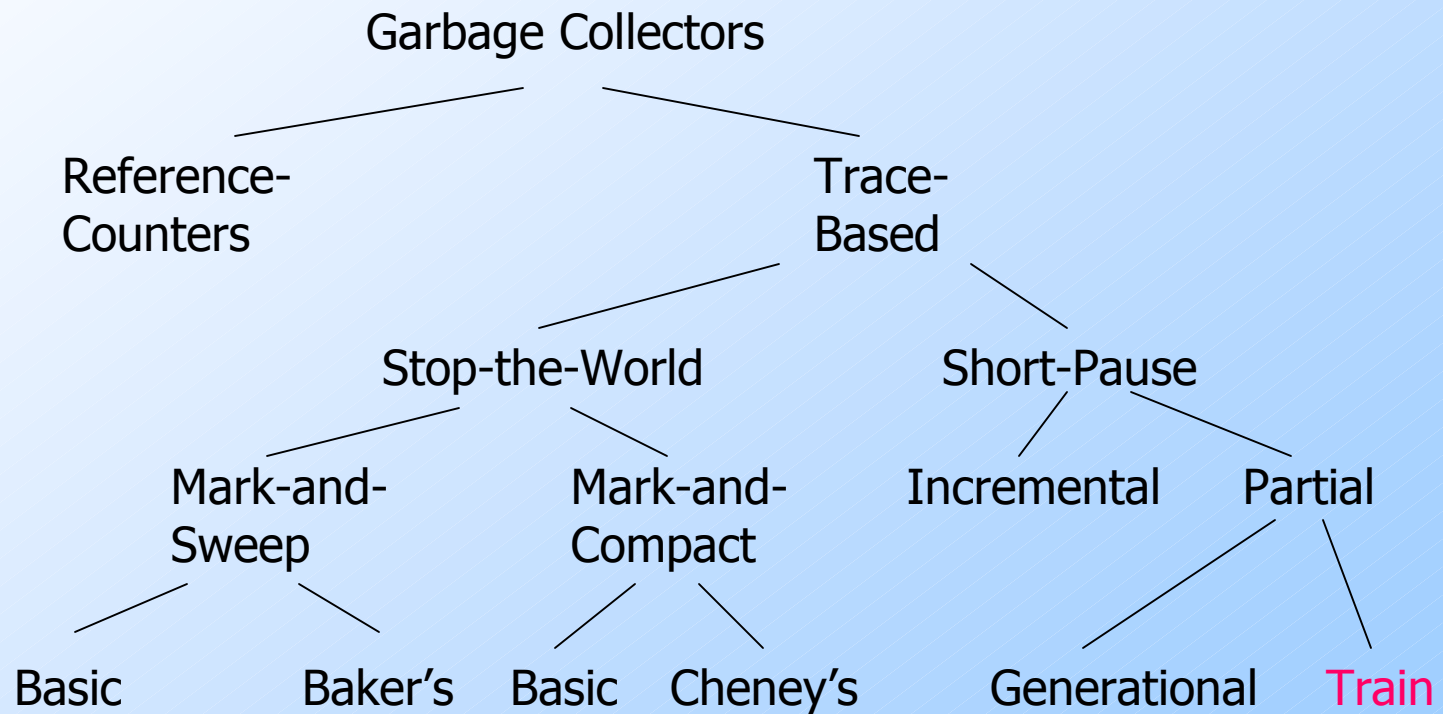
# Generational Garbage Collection

- ◆ Divide the heap into partitions  $P_0, P_1, \dots$ 
  - ◆ Each partition holds older objects than the one before it.
- ◆ Create new objects in  $P_0$ , until it fills up.
- ◆ Garbage collect  $P_0$  only, and move the reachable objects to  $P_1$ .

# Generational GC --- (2)

- ◆ When  $P_1$  fills, garbage collect  $P_0$  and  $P_1$ , and put the reachable objects in  $P_2$ .
- ◆ **In general:** When  $P_i$  fills, collect  $P_0, P_1, \dots, P_i$  and put the reachable objects in  $P(i+1)$ .

# Taxonomy

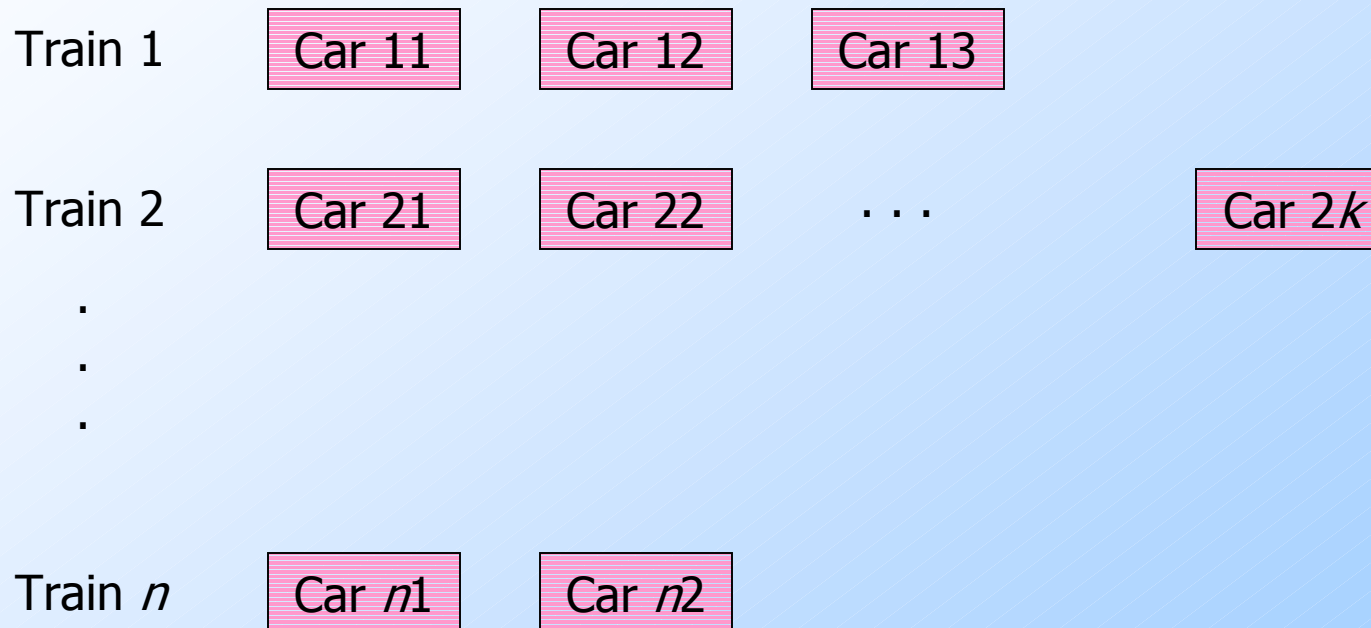




# The Train Algorithm

- ◆ Problem with generational GC:
  1. Occasional total collection (last partition).
  2. Long-lived objects move many times.
- ◆ Train algorithm useful for long-lived objects.
  - ◆ Replaces the higher-numbered partitions in generational GC.

# Partitions = "Cars"



# Organization of Heap

- ◆ There can be any number of trains, and each train can have any number of cars.
  - ◆ You need to decide on a policy that gives a reasonable number of each.
- ◆ New objects can be placed in last car of last train, or start a new car or even a new train.

# Garbage-Collection Steps

1. Collect the first car of the first train.
2. Collect the entire first train if there are no references from the root set or other trains.
  - ◆ **Important:** this is how we find and eliminate large, cyclic garbage structures.

# Remembered Sets

- ◆ Each car has a remembered set of references from later trains and later cars of the same train.
- ◆ **Important:** since we only collect first cars and trains, we never need to worry about “forward” references (to later trains or later cars of the same train).

# Collecting the First Car of the First Train

- ◆ Do a partial collection as before, using every other car/train as the stable set.
- ◆ Move all **Reachable** objects of the first car somewhere else.
- ◆ Get rid of the car.

# Moving Reachable Objects

- ◆ If object **o** has a reference from another train, pick one such train and move **o** to that train.
  - ◆ Same car as reference, if possible, else make new car.
- ◆ If references only from root set or first train, move **o** to another car of first train, or create new car.

# Panic Mode

- ◆ **The problem:** it is possible that when collecting the first car, nothing is garbage.
- ◆ We then have to create a new car of the first train that is essentially the same as the old first car.



# Panic Mode --- (2)

- ◆ If that happens, we go into *panic mode*, which requires that:
  1. If a reference to any object in the first train is rewritten, we make the new reference a “dummy” member of the root set.
  2. During GC, if we encounter a reference from the “root set,” we move the referenced object to another train.

## Panic Mode --- (3)

- ◆ **Subtle point:** If references to the first train never mutate, eventually all reachable objects will be sucked out of the first train, leaving cyclic garbage.
- ◆ But perversely, the last reference to a first-train object could move around so it is never to the first car.