Scope and Lifetime

• The **scope** of a declaration is portion of program text to which it applies (is **visible**).
  - Need not be a contiguous region.
  - In Java, as in Python, it is static: independent of data.

• The **lifetime** or **extent** of storage is the portion of program execution during which it exists.
  - Always contiguous.
  - Generally dynamic: depends on data

• Classes of extent:
  - **Static**: entire duration of program
  - **Local** or **automatic**: duration of the execution of a call or block—as for local variables or parameters.
  - **Dynamic**: From time of explicit allocation (**new**) to deallocation, if any.
Explicit vs. Automatic Freeing

- Java has no explicit means to free dynamic storage.
- However, when no expression in any thread can possibly be influenced by or change an object, it might as well not exist:

  ```java
  IntList wasteful() {
      IntList c = new IntList(3, new IntList(4, null));
      return c.tail;
      // variable c now deallocated, so no way
      // to get to first cell of list
  }
  
  At this point, Java's runtime, like Scheme's, "recycles" the object that c pointed to: garbage collection.
  ```
Under the Hood: Allocation

- Java pointers (references) are represented as integer addresses.
- Corresponds to machine’s own practice.
- In Java, cannot convert integers ↔ pointers,
- But crucial parts of Java’s runtime are implemented in C, or sometimes machine code, where you can conflate integers and pointers.
- Example of a crude allocator in C:

```c
char store[STORAGE_SIZE];   // Allocated array
size_t remainder = STORAGE_SIZE;

/** A pointer to a block of at least N bytes of storage */
void* simpleAlloc(size_t n) {   // void*: pointer to anything
    if (n > remainder) ERROR();
    remainder = (remainder - n) & ~0x7;   // Make multiple of 8
    return (void*) (store + remainder);
}
```
Example of Storage Layout: Unix

- OS provides a way to turn chunks of unallocated region into heap.
- Happens automatically for stack.
Explicit Deallocation

- C/C++ normally require explicit deallocation, because of
  - Lack of run-time information about types and array sizes;
  - Possibility of converting pointers to integers;
  - Lack of run-time information about unions:

    ```
    union Various {
        int Int;
        char* Pntr;
        double Double;
    } X;  // X is either an int, char*, or double
    ```

- Java avoids all three problems; automatic collection possible.

- Explicit freeing can be somewhat faster, but rather error-prone:
  - Memory corruption (freeing twice, freeing something that isn’t actually a valid pointer.)
  - Memory leaks (failing to ever release something.)
Free Lists

- Explicit allocator grabs chunks of storage from OS to give to applications.
- Or gives recycled storage, when available.
- When storage is freed, it is added to a **free list** data structure to be recycled.
- Used both for explicit freeing and some kinds of automatic storage management.

```
Variables (visible to program)

The Heap

Free List

Free list

In use
```

Last modified: Mon Apr 25 11:33:45 2022
Free List Strategies

- Memory requests generally come in multiple sizes.
- Not all chunks on the free list are big enough, and one may have to search for a chunk and break it up if too big.
- Various strategies to find a chunk that fits have been used:
  - **Sequential fits:**
    - Link blocks in LIFO or FIFO order, or sorted by address.
    - Coalesce adjacent blocks.
    - Search for *first fit* on list, *best fit* on list, or *next fit* on list after last-chosen chunk.
  - **Segregated fits:** separate free lists for different chunk sizes.
  - **Buddy systems:** A kind of segregated fit where some newly adjacent free blocks of one size are easily detected and combined into bigger chunks.
- Coalescing blocks reduces *fragmentation* of memory into lots of little scattered chunks.
Automatic Garbage Collection: Reference Counting

- Idea: Keep count of number of pointers to each object. Release when count goes to 0.

```
      Y:  □  

      X:  □  →  □  →  □  →  □  
             □  →  □  →  □  
                 □  →  □  

      1  □  →  1  □  →  1  □  →  1  □  
            □  →  □  →  □  
                □  →  □  

      X = Y;  

      Y:  □  

      X:  □  →  □  →  □  →  □  
             □  →  □  →  □  
                 □  →  □  

      0  □  →  3  □  →  1  □  →  1  □  
            □  →  □  →  □  
                □  →  □  

      Y = X.tail;  

      Y:  □  

      X:  □  →  □  →  □  →  □  
             □  →  □  →  □  
                 □  →  □  

      1  □  →  2  □  →  1  □  →  1  □  
            □  →  □  →  □  
                □  →  □  

      X = Y;  

      Y:  □  

      X:  □  →  □  →  □  →  □  
             □  →  □  →  □  
                 □  →  □  

      0  □  →  2  □  →  1  □  →  1  □  
            □  →  □  →  □  
                □  →  □  

...etc., until:
```
Garbage Collection: Mark and Sweep

Roots (locals + statics)

1. Traverse and mark graph of objects.
2. Sweep through memory, freeing unmarked objects.

Before sweep:

<table>
<thead>
<tr>
<th>A</th>
<th>B*</th>
<th>C</th>
<th>D*</th>
<th>E*</th>
<th>F</th>
<th>G*</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>D</td>
<td>G</td>
<td>F</td>
<td>A</td>
<td>7</td>
<td>G</td>
</tr>
</tbody>
</table>

After sweep:

<table>
<thead>
<tr>
<th>B</th>
<th>D</th>
<th>G</th>
<th>D</th>
<th>E</th>
<th>G</th>
</tr>
</thead>
</table>
Cost of Mark-and-Sweep

- Mark-and-sweep algorithms don’t move any existing objects—pointers stay the same.

- The total amount of work depends on the amount of memory swept—i.e., the total amount of active (non-garbage) storage + amount of garbage. Not necessarily a big hit: the garbage had to be active at one time, and hence there was always some “good” processing in the past for each byte of garbage scanned.
Copying Garbage Collection

- Another approach: *copying garbage collection* takes time proportional to amount of active storage:
  - Traverse the graph of active objects breadth first, *copying* them into a large contiguous area (called “to-space”).
  - As you copy each object, mark it and put a *forwarding pointer* into it that points to where you copied it.
  - The next time you have to copy an already marked object, just use its forwarding pointer instead.
  - When done, the space you copied from (“from-space”) becomes the next to-space; in effect, all its objects are freed in constant time.
Copying Garbage Collection Illustrated

(a) Roots from: 42 D G F A 7 G D C E to:

(b) Roots from: 42 B' G F A 7 G E' C E to:

(c) Roots from: 42 B' G F A D' 7 G E' C G' E to:

(d) Roots from: 42 B' G F A D' 7 G E' C G' E to:

B: Old object
B': New object
*: marked

Forwarding pointers
Copy roots
Copy from to-space in (b).
Only D is new
Copy from to-space in (c).
No new objects
Most Objects Die Young: Generational Collection

- Most older objects stay active, and need not be collected.
- Would be nice to avoid copying them over and over.

*Generational garbage collection* schemes have two (or more) from spaces: one for newly created objects (*new space*) and one for “tenured” objects that have survived garbage collection (*old space*).

- A typical garbage collection collects only in new space, ignores pointers from new to old space, and moves objects to old space.
- As roots, uses usual roots plus pointers in old space that have changed (so that they might be pointing to new space).
- When old space full, collect all spaces.
- This approach leads to much smaller *pause times* in interactive systems.
There's Much More

- These are just highlights.
- Lots of work on how to implement these ideas efficiently.
- **Distributed garbage collection:** What if objects scattered over many machines?
- **Real-time collection:** where predictable pause times are important, leads to **incremental** collection, doing a little at a time.