RUN-TIME ENVIRONMENTS

CMPT 379 Lecture 23a

Lecture Overview

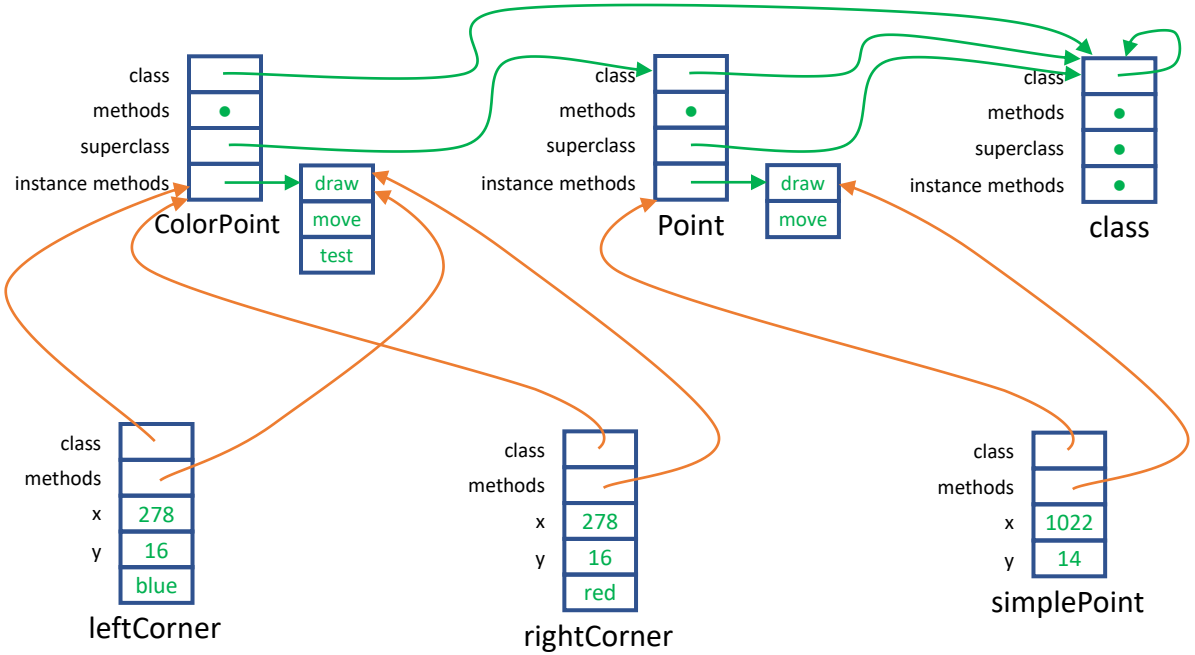
- Language Feature Support
  - Objects
  - Interoperability
- Operating System Interface
  - Standard Objects
  - Run-time Libraries
- Memory Management
  - Frame Stack Management
  - Heap Management

Chapter 6

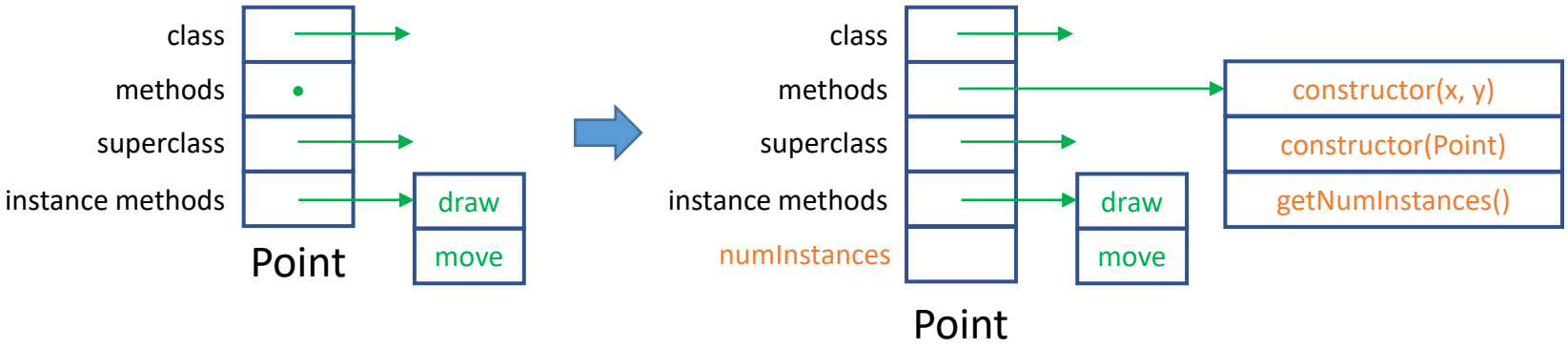
Language Feature Support

- Most languages have features which require some sort of support at run-time.
- This support is particular to the language and the compiler implementation, but generally not to the program being compiled.
- Example features that need support are objects, interoperability with other languages, closures, dynamic name spaces, co-routines and other parallelism, an interface with the operating system, and memory management.

Support for Objects



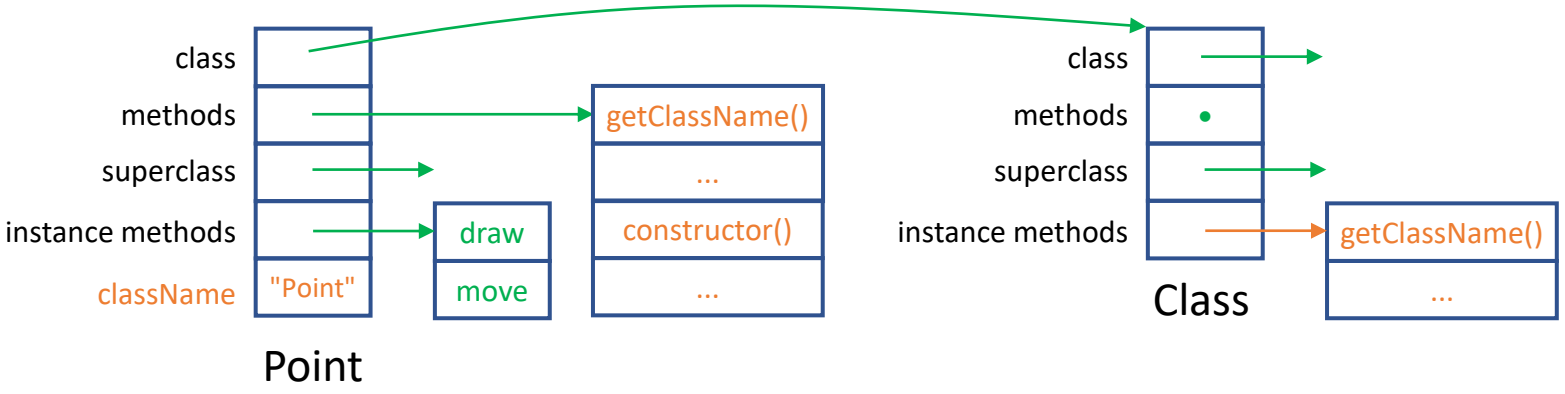
Static Members for Objects



Static members (fields and methods) can be added to the class object. This includes constructors.

Class-level Methods

Some languages have methods that apply to any class. Reflection methods in java are an example.



Interoperability

- Many languages allow for interoperability between that language and other languages.
- Examples are the Java Native Interface (JNI) and the .NET framework.
- Interoperability concerns can be broadly classified as:
  - data layout
  - procedure calling (e.g. using standardized linkages)
  - run-time support for the other language (e.g. memory management, operating system interface)

Operating System Interface

- Almost all programs will need to interact with the operating system in some way.
- This includes reading input, writing output, opening, reading, writing, and closing files or sockets, getting fresh pages of memory, changing process priority, spawning or ending processes, throwing hardware-related exceptions, etc.
- Each operating system has different interfaces to connect to, so this is often done on a compiler-to-operating-system basis.

Operating System Interface

- Code that interfaces between a language and an operating system is often incorporated as a library and/or as a collection of standard objects.
- For example, C uses the standard C runtime library (-libcrt) and java has the System object and its subobjects (like System.out).
- The compiler often must make provision for these libraries or objects, because they use features that are generally not available in the source language. They may, for instance, use traps.

Memory Management

- Programs in modern languages have varying needs in terms of managing the memory space.
- Typically, the language, compiler, and operating system run some sort of memory management in the background.
- Data layout is one area where the program still sometimes makes a difference. A C struct or array specifies the layout of the data in memory.
- In other languages, for instance C#, struct or object records can change the order of their fields.

Frame Stack Maintenance

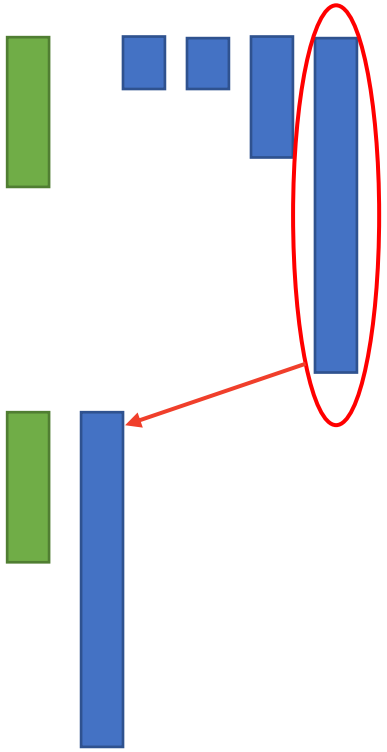
- The heap and the frame stack are two places where the language and compiler do most of the management work.
- For the frame stack, the compiler needs to insert standardized linkages that create and destroy the frames on the stack, and have other conventions (around argument passing, for instance) incorporated into the code.
- The compiler also needs to generate and maintain a display if that is a feature.

Heap Maintenance

- For the heap, the amount of work the compiler must do varies.
- Many languages support explicit memory allocation.
- Many languages have implicit memory allocation that happens when an object is created.
- Those that do must maintain a list of free blocks—blocks of memory that the program isn't currently using.

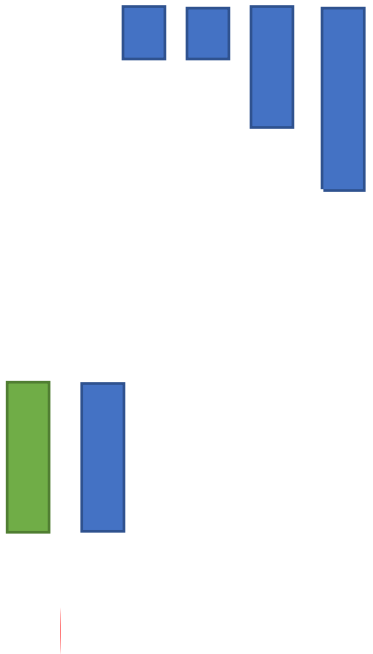
Allocation

- Allocating memory from the heap is generally a simple process.
- The heap manager looks at the list of free blocks that it has and finds one whose size is at least as big as the request (plus overhead for some header information).
- If no block is big enough, a new block is made.
- This block is then split (if necessary or desirable) to be the size required, with split-off parts placed in the free block list.



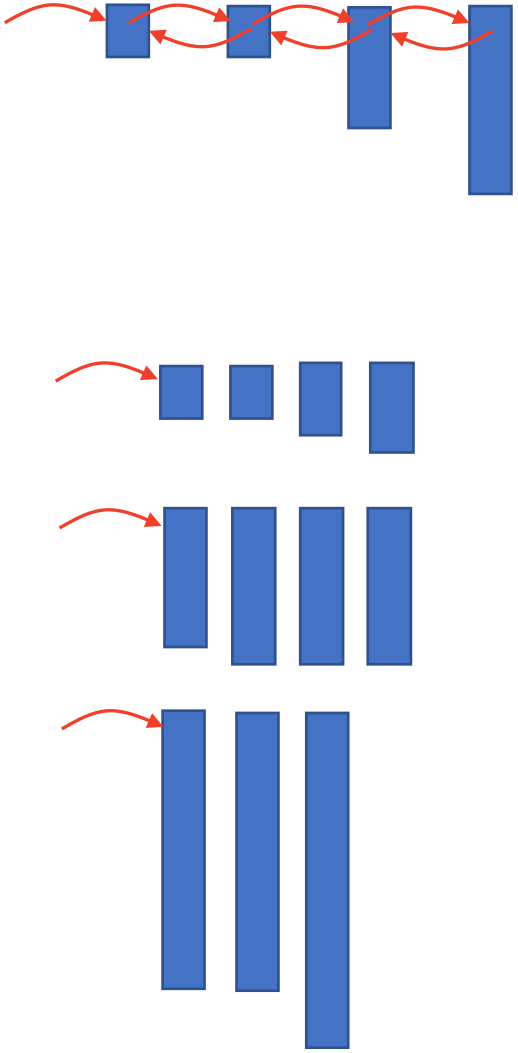
Allocation

- Allocating memory from the heap is generally a simple process.
- The heap manager looks at the list of free blocks that it has and finds one whose size is at least as big as the request (plus overhead for some header information).
- If no block is big enough, a new block is made.
- This block is then split (if necessary or desirable) to be the size required, with split-off parts placed in the free block list.



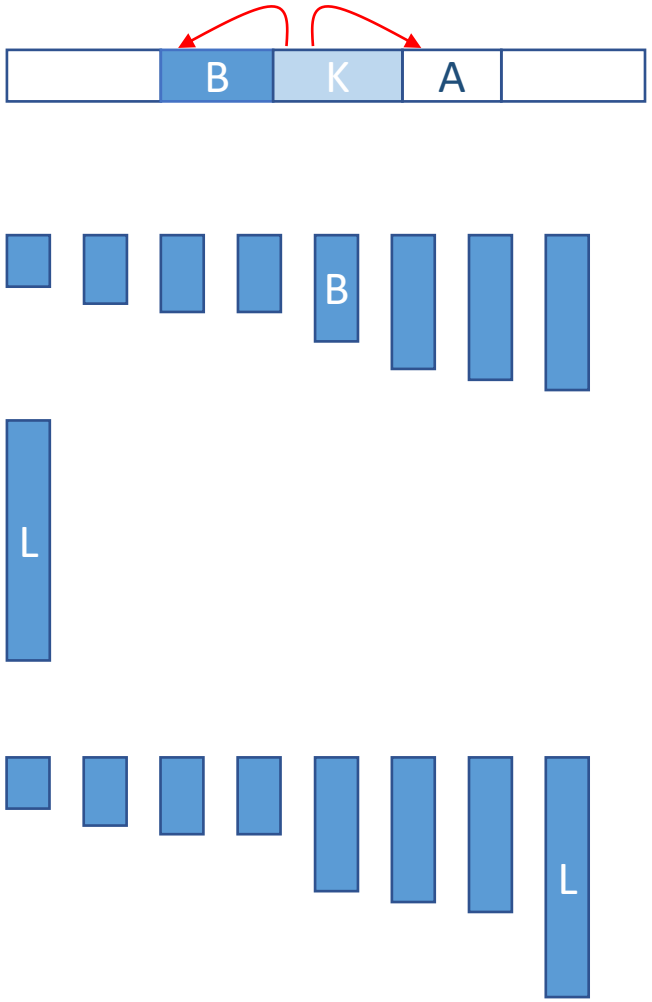
Free Blocks

- Each block of memory (free or used) has header information in it.
- Free blocks can be held in a doubly-linked list, where the pointers are in the headers.
- Headers may also contain pointers to the previous and the next block in memory. This helps consolidate blocks during deallocation.
- Headers can also contain status information, such as a "in use" bit and a size.
- Often, the free block "list" is a set of lists of blocks of different sizes.



Deallocation of a Block

- When a block K is deallocated, check to see if the block B before and/or the block A after it in memory is in use.
- If not, then take whichever of A and B are not used from the free list and merge it with K. Put the resulting block on the free list.



When Should a Block be Deallocated?

- Frames are easy – they are deallocated when the procedure ends.
- Heap memory is more complex. There are two main approaches:
  - Explicit Deallocation: here the source language programmer must tell the heap manager when a used block becomes free.
  - Implicit Deallocation: the heap manager automatically detects when a block is free. This can be done by keeping reference counts to blocks or by periodically scanning memory for unused blocks (called garbage collection).
- Explicit deallocation is faster but puts more of a burden on the source programmer. In complex systems it is often hard to tell when a block of memory is no longer needed. It allows for entire classes of bugs that implicit deallocation prevents.

Reference Counts

- In reference counting, each block of memory keeps a count of how many references (pointers) there are to that block.
- Whenever a pointer is changed, the block the pointer was pointing at has its count decreased, and the block the pointer will now point to has its count increased.
- Counts are decreased when pointers go out of scope also.
- If a block's count reaches zero, then at some appropriate time (say, the end of the statement causing the count to go to zero) the block is deallocated.
- Circularly linked structures could become a problem.

Garbage Collection

- In garbage collection, the heap manager periodically (either at fixed intervals or when memory is needed) scans through memory looking for unused blocks.
- This can cause programs to stop for a long time while a lot of memory is searched.
- Some systems do partial scans at more frequent intervals so there are less noticeable pauses.
- The "mark and sweep" method is the most popular.

Mark and Sweep

- In mark and sweep, the heap manager follows all the pointers in the program to find all the used blocks.
  - marks every block in memory as unused.
  - starts with static memory:
    - for every pointer, the block that it is pointing to is marked as used.
    - used blocks are searched for pointers, and the blocks those pointers point at are marked as used (recursively).
  - continues with the frames in the frame stack
    - again, every pointer has its block recursively marked as used.
  - visits every block in memory, collecting those that are marked unused into a free list.
- This relies on knowing or detecting where the pointers are in every type of block and frame, and in static memory.