

CMPE 152: Compiler Design

November 28 Class Meeting

Department of Computer Engineering
San Jose State University



Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak



Extra-Credit Oral Presentations

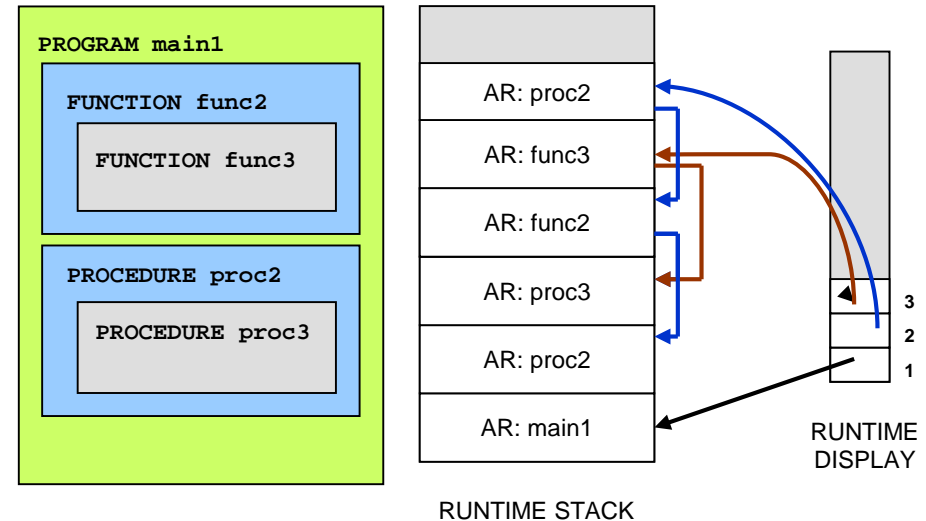
- ❑ Let me know if your team would like to do an oral presentation for extra credit.
- ❑ Tell about your language.
 - Show some example programs.
- ❑ Describe its grammar
 - Show syntax diagrams.
- ❑ What Jasmin code do you generate?
 - Show some code diagrams.
- ❑ Demo: Compile, execute, and run some sample programs.

Extra-Credit Oral Presentations, *cont'd*

- ❑ Submit a note by Friday, Dec. 1 into Canvas: [Assignments | Miscellaneous | Presentation](#) if your team wants to present.
 - Choose either Dec. 5 or 7 to present.
- ❑ 15-20 minutes.
- ❑ Can add up to 50 points to each team member's total assignment score.

Static Scoping

- ❑ Pascal uses **static (lexical) scoping**.
- ❑ Scopes are determined at compile time by how the routines are nested in the source program.



main1 → proc2 → proc3 → func2 → func3 → proc2

- ❑ At runtime, variables are “**bound**” to their values as determined by the lexical scopes.
- ❑ The runtime display helps do the bindings.

Static Scoping in Java

- What value is returned by calling function `g()`?

```
int x = 0;  
int f() { return x; }  
int g() { int x = 1; return f(); }
```

What is the binding
of this `x` with
static scoping?

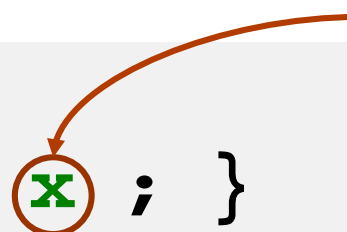
Dynamic Scoping

- ❑ With **dynamic scoping**, runtime binding of variables to their values is determined by the call chain.
- ❑ To determine a variable's binding, search the call chain backwards starting with the currently active routine.
- ❑ The variable is bound to the value of the first declared variable found with the same name.

Hypothetical Dynamic Scoping in Java

- What value is returned by calling function `g()`?

```
int x = 0;  
int f() { return x; }  
int g() { int x = 1; return f(); }
```



What is the binding
of this `x` with
dynamic scoping?

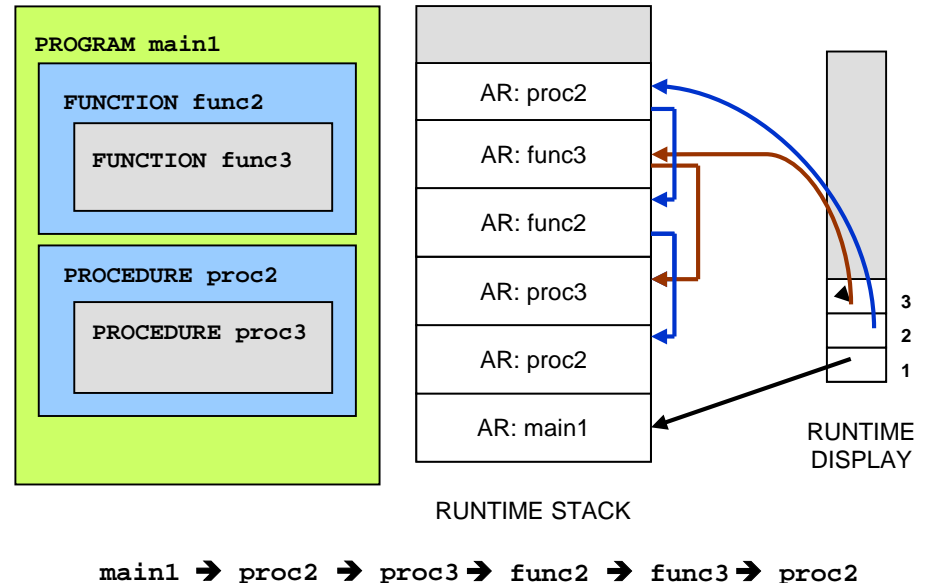
- Call chain: `main` → `g` → `f`

How would you implement
runtime dynamic scoping?

Runtime Memory Management

- In the “Pascal Virtual Machine”, all local data is kept on the runtime stack.

- All memory for the parameters and variables declared locally by a routine is allocated in the routine's activation record.



- The memory is later automatically deallocated when the activation record is popped off the stack

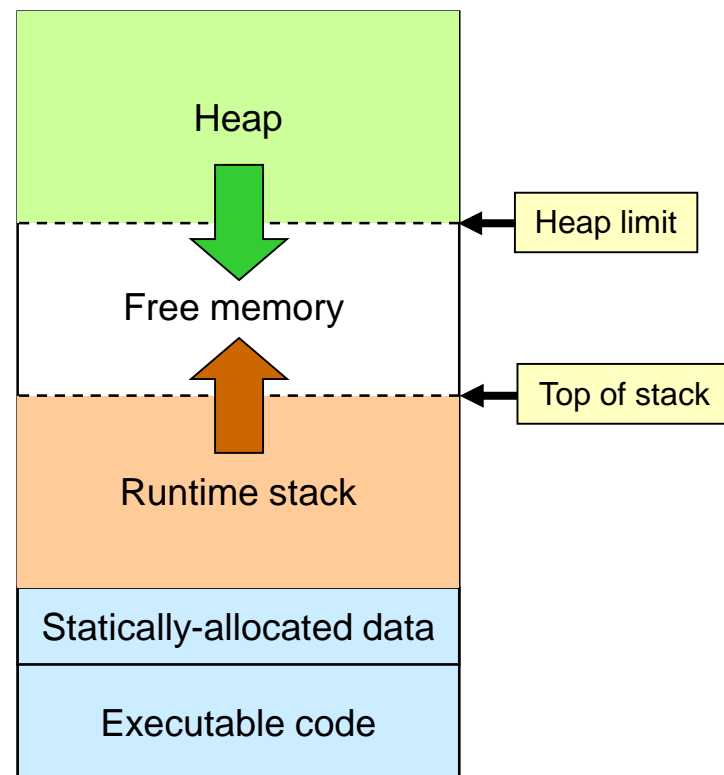
Runtime Memory Management, *cont'd*

- What about dynamically allocated data?
- Memory for dynamically allocated data is kept in the “heap”.

Runtime Memory Management, *cont'd*

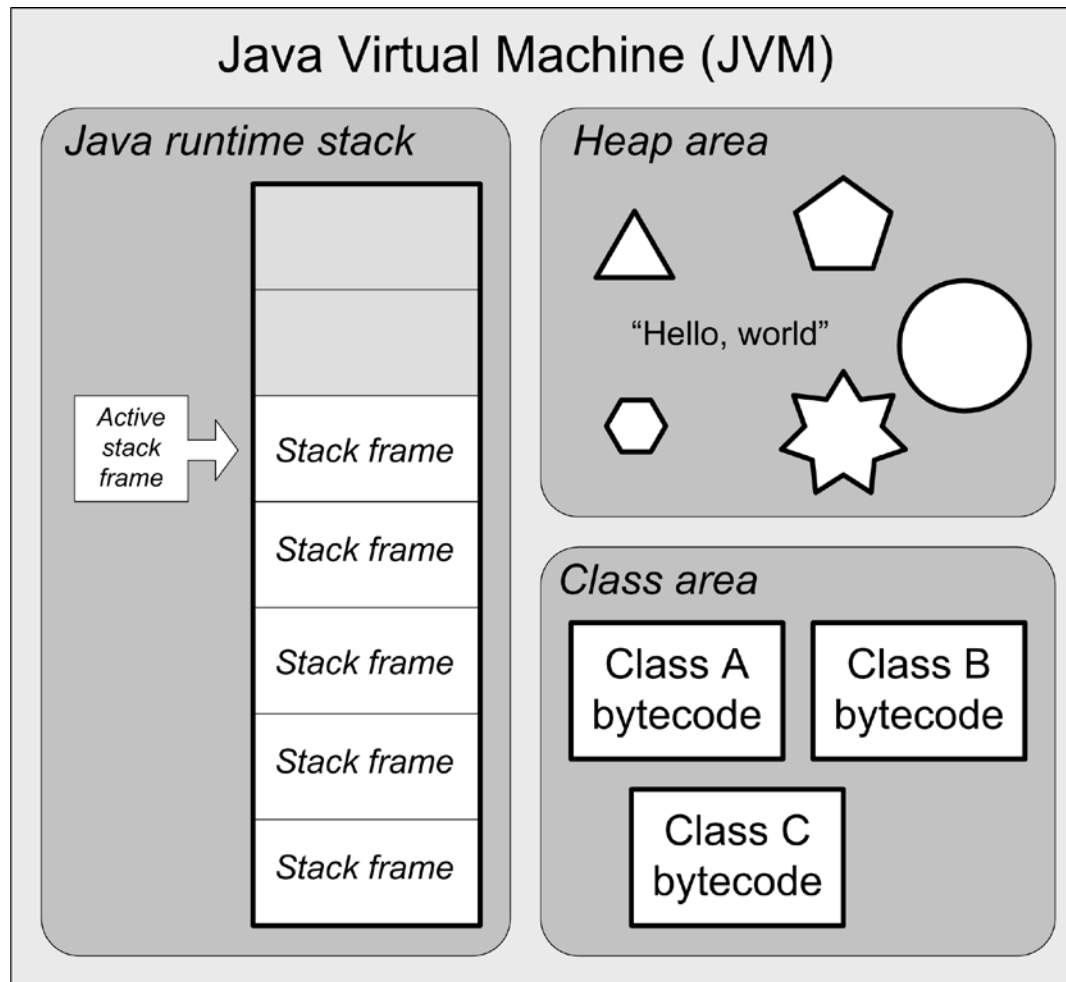
□ Runtime memory can be divided into four partitions:

1. Static memory
 - executable code
 - statically-allocated data
2. Runtime stack
 - activation records that contain locally-scoped data
3. Heap
 - dynamically-allocated data
 - such as Java objects
4. Free memory



Avoid: Heap-stack collision
(Out of memory error)

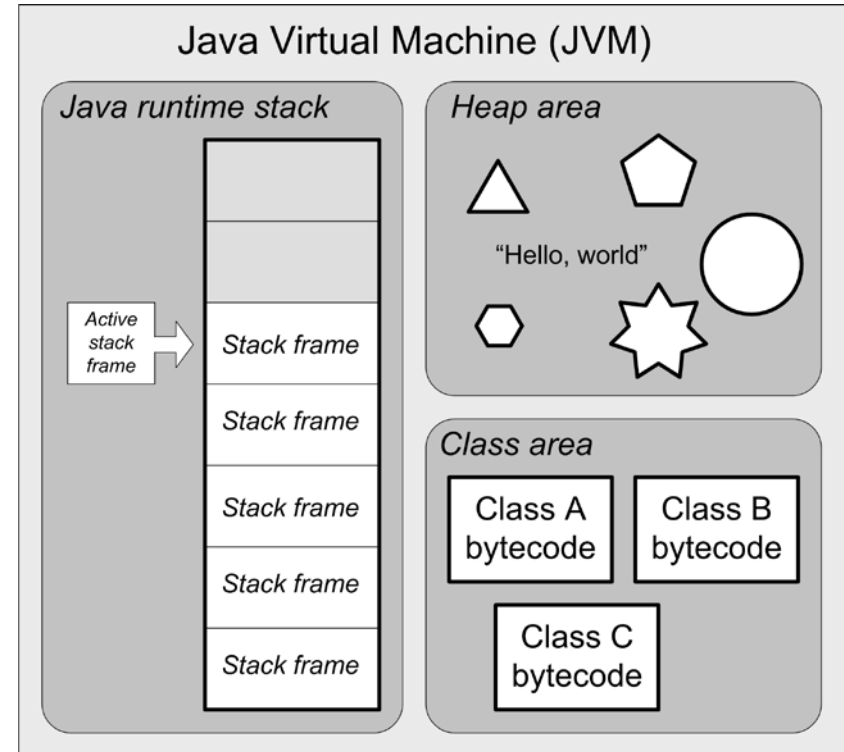
Recall the JVM Architecture ...



Java Command-Line Arguments

□ **java -Xms<size> -Xmx<size>**

- **ms**: initial heap size
- **mx**: maximum heap size
- **<size>**: size in megabytes (**m|M**) or gigabytes (**g|G**)



- Example: **java -Xms512M -Xmx2G HelloWorld**

Runtime Heap Management

- Handled by language-specific runtime routines.
- Pascal, C, and C++
 - Call **new/malloc** to allocate memory.
 - Call **dispose/free** to de-allocate.
- Java
 - Call **new** to allocate memory.
 - Set the initial and maximum heap size using the **-Xms** and **-Xmx** options.
 - **Automatic garbage collection.**

Runtime Heap Management, *cont'd*

- ❑ Keep track of all allocated blocks of memory and all free blocks.
- ❑ Free blocks are “holes” caused by freeing some of the dynamically allocated objects.

Runtime Heap Management, *cont'd*

- ❑ When a new block of memory needs to be allocated dynamically, where should you put it?
- ❑ You can allocate it at the end of the heap, and thereby expand the size of the heap.
- ❑ You can find a hole within the heap that's big enough for the block.

Runtime Heap Management, *cont'd*

- What's the optimal memory allocation strategy?
 - Find the smallest possible hole that the block will fit in.
 - Randomly pick any hole that's big enough.
- Should you periodically compact the heap to get rid of holes and thereby reduce the size of the heap?
- If allocated data moves due to compaction, how do you update references (pointers) to the data?

Garbage Collection

- Return a block of memory (“garbage”) to unallocated status ...
 - ... when there are no longer any references to it.
- Various algorithms to locate garbage.
 - reference counts
 - mark and sweep
 - stop and copy
 - generational

Automatic Garbage Collection

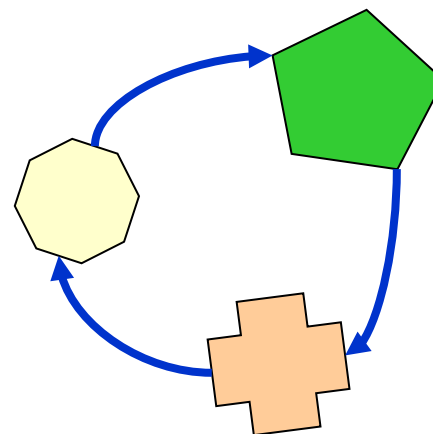
- Automatic garbage collection is great for programmers, because you don't have to think about it.
- But it can slow runtime performance unpredictably.
 - How?

Garbage Collection: Reference Counts

- Include a counter with each block of allocated memory.
 - Increment the counter each time a pointer is set to point to the block.
 - Decrement the counter whenever a pointer to the block is set to null (or to point elsewhere).
 - Deallocate the block when the counter reaches 0.

- Problem: **Cyclic graphs**

- The reference counts never become 0.



Garbage Collection: Mark and Sweep

- Make a pass over the heap to **mark** all allocated blocks of memory that are reachable via pointers.
 - Various marking algorithms.
- Make a second pass to **sweep** (deallocate) blocks that are not marked.

Garbage Collection: Stop and Copy

- Partition the heap into two halves.
 - Allocate memory only in one half at a time.
- When an allocation fails to find a sufficiently large block of free memory ...
 - Stop
 - Copy all allocated blocks to the other half of the heap, thereby compacting it.
 - Update all references to the allocated blocks.
 - How do you update pointer values?

Generational Garbage Collection: Theory

- Most objects are short-lived.
- Long-lived object are likely to last until the end of the run of the program,

Generational Garbage Collection: Practice

- Partition the heap into a **new generation** area and an **old generation** area.
 - Allocate new objects in the new generation area.
 - Keep track of how long an object lives.
 - Once an object lives past a predetermined threshold of time, migrate it to the old generation area.
- The old generation area stays fairly compacted.
- The new generation area needs compacting infrequently.

Aggressive Heap Management

- Aggressive heap management means doing garbage collection frequently, even when it's not necessary.
 - There's still adequate free space remaining in the heap area.
- Keep the heap as small as possible.
 - Improve reference locality.
 - Optimize the use of physical memory when multiple programs are running.
 - Reduce virtual memory paging.

Aggressive Heap Management: Tradeoff

- ❑ GC operations slow program performance.
- ❑ But paging to disk can be orders of magnitude slower.

Garbage Collection Research

- ❑ Entire books have been written about garbage collection.
- ❑ It's still an area with opportunities for research.
- ❑ You can become famous by inventing a better GC algorithm!
- ❑ Maybe some form of **adaptive GC** using machine learning?