Comp 204: Computer Systems and Their Implementation

Lecture 23: Java Development and Run-Time Store Organisation

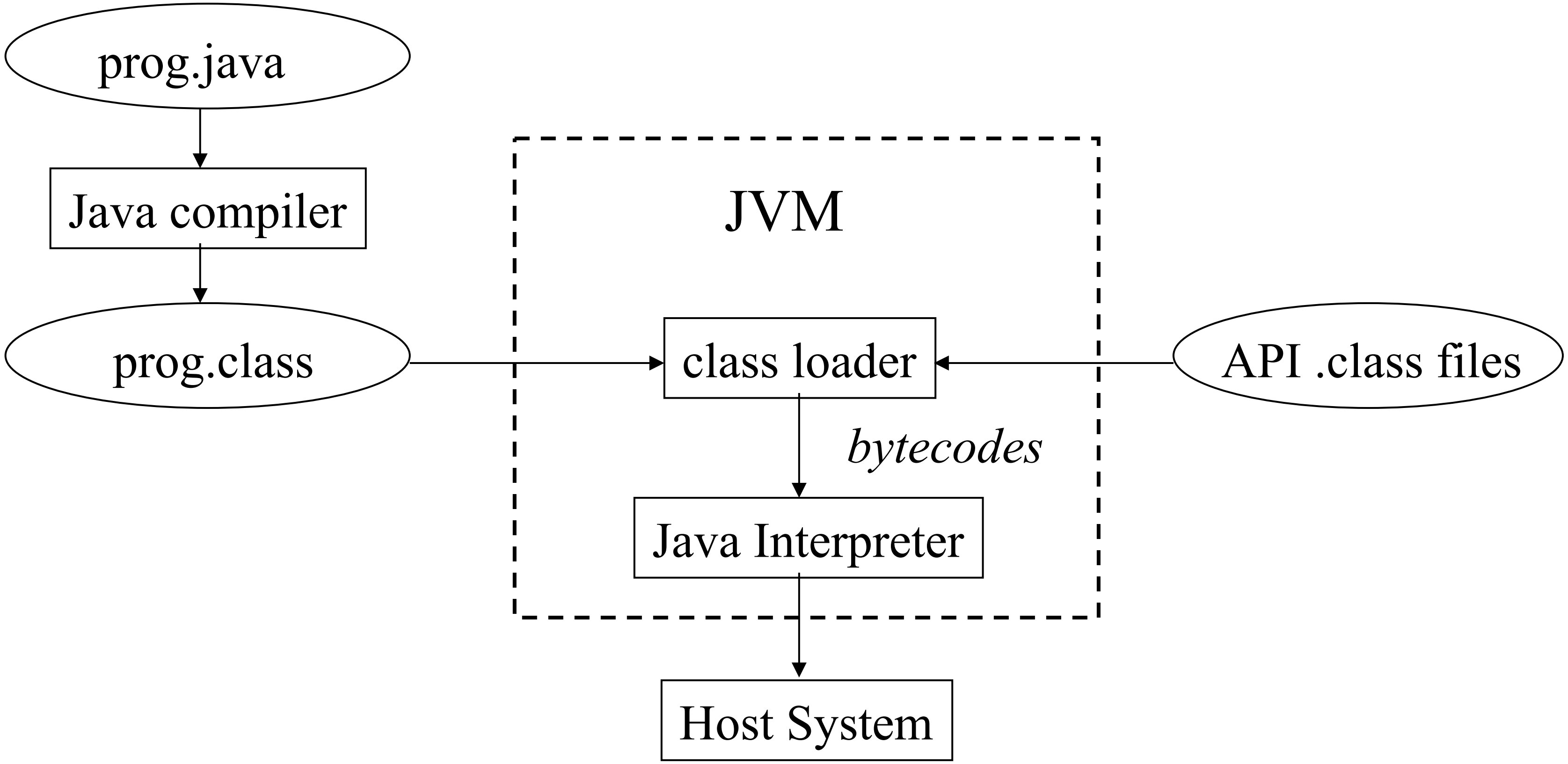
Today

- Java development
- Run-time store organisation
  - Data frames
  - Heap storage

The Java Approach

- Java programs are intended to be architecture neutral
- Portability is achieved by creating a Java platform on each architecture
- The Java platform consists of
  - a Java Virtual Machine (JVM)
  - an Application Programming Interface (API)

Java Development Environment



Java Development

- The API enables the programmer to communicate with and control the host environment
  - it provides support for I/O, graphics, networking and various utilities
- The Java compiler generates virtual machine code called bytecode
  - this can be run on any Java platform
- When a Java program or applet is run, an instance of the JVM is created

JVM

- The class loader links the program bytecode with API bytecode files
- Java interpreter then performs actions specified by bytecodes
- For increased efficiency...
  - interpreter may be implemented in hardware/ microcode
  - interpreter may be replaced by a JIT (Just In Time) compiler
    - converts bytecode to native machine code just prior to execution

Run-Time Store Organisation

- As well as generating code, the compiler must allocate memory to hold declared objects

- The area of memory used to hold variables etc. for a particular subprogram is a data frame (or stack frame or activation record)

- For most modern languages, need to allocate data frames dynamically
  - usual solution is a stack

Question

- Why can't we allocate data frames statically, i.e. have one fixed area for each subprogram? Which of the following are true

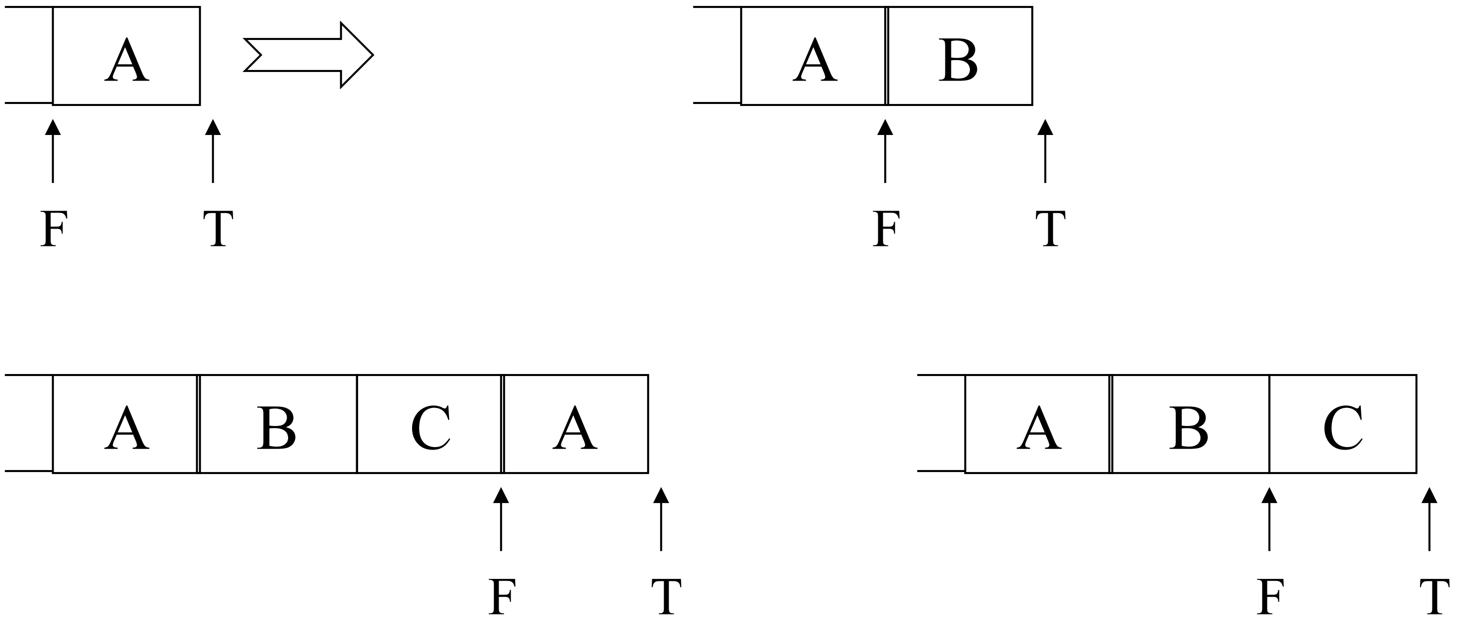
I. Data Structures may be dynamically allocated
II. Object Orientation demands the creation of Instances
III. Recursion causes data frames to grow arbitrarily

a) I only
b) III only
c) I and II only
d) II and III only
e) I, II and III only

Answer: c
I and II only; recursion does not affect the size of data frames

A calls B calls C calls A

- F = Frame pointer T = Top of stack



Data Frame

Control info.	Parameters	Local variables

- Control information includes
  - R = Return address of calling subprogram
  - P = Start address of previous frame

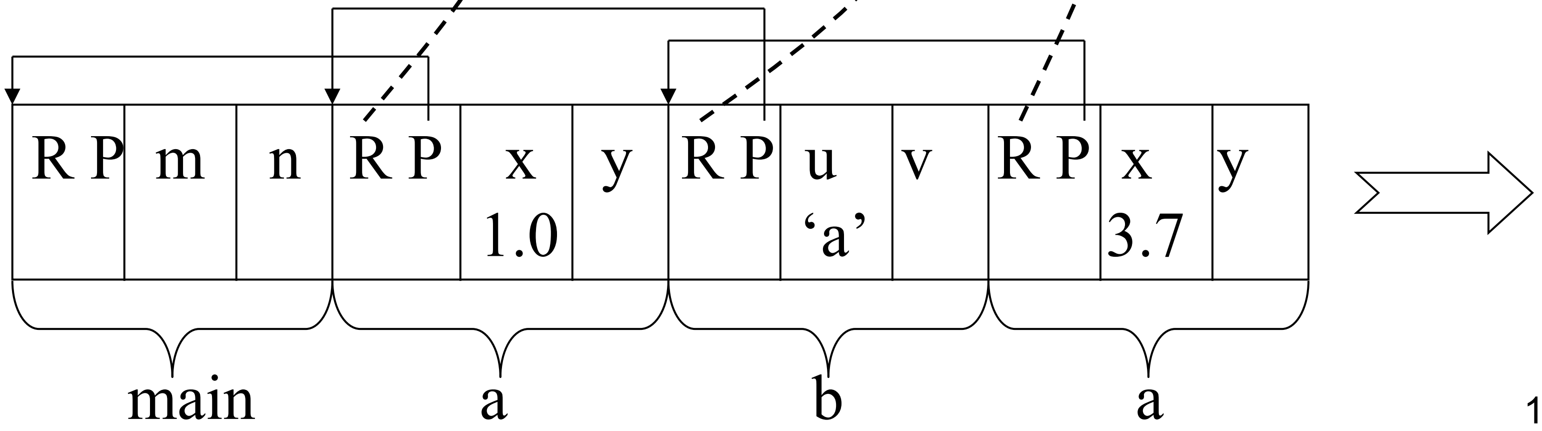
- Some languages allow run-time sizing of objects
  - may have to grow frame dynamically

- Data items accessed as offset from frame pointer
  - e.g. LOAD 1, 3(F)

```
void a (float x) {
   float y;
   ...
   b('a');
   ...
}
```

```
void b (char u) {
   boolean v;
   ...
   a(3.7);
   ...
}
```

```
void main(...) {
  int m, n;
  a(1.0)
  ...
}
```



Question

- Which of the following is usually NOT represented in a subroutine's activation record frame for a stack-based programming language?

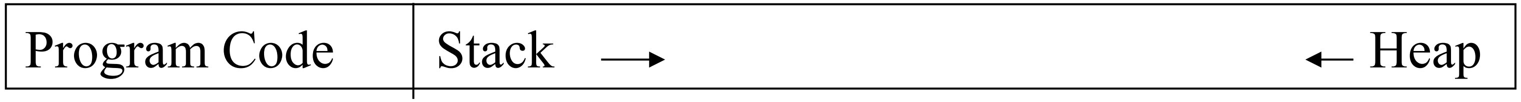
  a) Values of locally declared variables
  b) A heap area
  c) The return address
  d) A pointer to the calling activation record
  e) Parameter values passed to the subroutine

Answer: b
A heap area

Heap Storage

- Sometimes, dynamically allocated storage has to remain available even when a subprogram terminates
  - e.g. list processing applications
  - object oriented code (*new*)

- Solution is to use a heap



Program Code | Stack → ← Heap

Heap Storage

- If heap grows too large, may have to do garbage collection
  - involves reclaiming those areas of heap no longer required/accessible
  - costly to do automatically (Java, LISP)
  - may be left to programmer
    - e.g. *free*() call in C