

Topic 10

Java Memory Management

1-1

Memory Allocation in Java

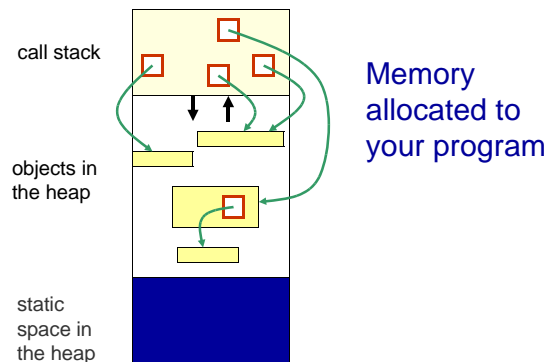
- When a program is being executed, separate areas of memory are allocated for each
 - class
 - interface
 - object
 - running method

1-2

Memory Allocation in Java

- **Call stack / runtime stack**
 - Used for *method* information *while the method is being executed*
 - Local variables
 - Formal parameters
 - Return value
 - Where method should return to
- **Heap**
 - Used for
 - **Static** information (interfaces and classes)
 - **Instance** information (objects)

1-3



1-4

Memory Allocation in Java

- **Example:** What happens when an object is created by **new**, as in `Person friend = new Person(...);`
 - The reference variable has memory allocated to it on the **call stack**
 - The object is created using memory in the **heap**

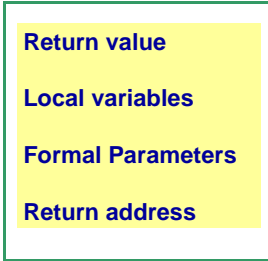
1-5

Runtime Stack

- **Call stack (runtime stack)** is the memory space used for *method* information *while a method is being run*
- When a method is invoked, a **call frame** (or **activation record**) for that method is created and "pushed" onto the call stack
 - All the information needed during the execution of the method is grouped together in the call frame

1-6

Call Frame (Activation Record) for a Method



1-7

Call Frame (Activation Record)

- A **call frame** contains:
 - Address to return to after method ends
 - Method's formal parameter variables
 - Method's local variables
 - Return value (if any)
- Note that the values in a call frame are accessible **only** while the corresponding method is being executed!

1-8

```
public class CallStackDemo
{
    public static void m2()
    {
        System.out.println("Starting m2");
        System.out.println("m2 calling m3");
        m3();
        System.out.println("m2 calling m4");
        m4();
        System.out.println("Leaving m2");
        return;
    }
    public static void m3()
    {
        System.out.println("Starting m3");
        System.out.println("Leaving m3");
        return;
    }
}
```

Example: a Typical Calling Sequence

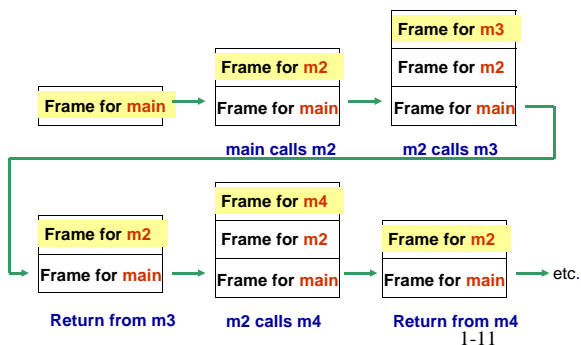
1-9

```
public static void m4()
{
    System.out.println("Starting m4");
    System.out.println("Leaving m4");
    return;
}

public static void main(String args[] )
{
    System.out.println("Starting main");
    System.out.println("main calling m2");
    m2();
    System.out.println("Leaving main");
}
```

1-10

Call Stack for a Typical Calling Sequence



1-11

Call Stack for a Typical Calling Sequence

- When the **main** method is invoked:
 - A **call frame** for **main** is created and pushed onto the runtime stack
- When **main** calls the method **m2**:
 - A **call frame** for **m2** is created and pushed onto the runtime stack
- When **m2** calls **m3**:
 - A **call frame** for **m3** is created and pushed onto the runtime stack
- When **m3** terminates, its call frame is popped off and control returns to **m2**

1-12

Call Stack for a Typical Calling Sequence

- When **m2** now calls **m4**:
 - What happens next?
 - What happens when **m4** terminates?
- What happens when **m2** terminates?
- What happens when **main** terminates?
Its call frame is popped off and control returns to the operating system

1-13

Call Frames

- We will now look at some examples of what is in a call frame for a method
 - First for simple variables
 - Then for reference variables

1-14

Example: Call Frames - Simple Variables

```
public class CallFrameDemo1
{
    public static double square(double n){
        double temp;
        temp = n * n;
        return temp;
    }

    public static void main(String args[] )    {
        double x = 4.5;
        double y;
        y = square(x);
        System.out.println("Square of " + x + " is " + y);
    }
}
```

Call Frames – Example 1

Draw a picture of the call frames on the call stack:

- What will be in the call frame for the **main** method?
 - Address to return to in operating system
 - Variable **args**
 - Variable **x**
 - Variable **y**
- What will be in the call frame for the method **square**?
 - Address to return to in main
 - Variable **n**
 - Variable **temp**
 - Return value

1-16

Discussion

- There will be a call frame on the call stack for **each** method called. So what other call frame(s) will be pushed onto the call stack for our example?
- Which call frames will be on the call stack at the same time?

1-17

Heap Space

- **Static space:** contains **one** copy of each class and interface named in the program
 - Contains their static variables, and methods
- **Object space:**
 - Information is stored about **each** object:
 - Value of its instance variables
 - Type of object (i.e. name of class)

1-18

Object Creation

- Now let's look at reference variables ...
- Memory is allocated in the **heap** area when an object is created using **new**
 - The reference variable is put in the **call frame** on the **runtime stack**
 - The object is created using memory in the **heap**

1-19

```
public class CallFrameDemo2 {  
    private static void printAll(String s1, String s2, String s3){  
        System.out.println(s1.toString());  
        System.out.println(s2.toString());  
        System.out.println(s3.toString());  
    }  
    public static void main(String args[] ) {  
        String str1, str2, str3;  
  
        str1 = new String(" string 1 ");  
        str2 = new String(" string 2 ");  
        str3 = new String(" string 3 ");  
  
        printAll(str1, str2, str3);  
    }  
}
```

Example: Call Frames-
Reference Variables

Call Frames – Example 2

Draw a picture of the call stack and of the heap as the program executes

- What will be the *sequence of call frames* on the call stack?

for **main**

for **String constructor** for **str1** – then popped off

for **String constructor** for **str2** – then popped off

for **String constructor** for **str3** – then popped off

for **printAll**

for **toString** for **str1** – then popped off

for **System.out.println** – then popped off

etc.

1-21

Call Frames – Example 2

- What will be in the call frame for **main**? (and in the heap?)
 - Address to return to in operating system
 - Variable **args**
 - Variable **str1**
 - Initially?
 - After return from **String constructor**?
 - Variable **str2**
 - Variable **str3**
- What will be in the call frame for **printAll**?

1-22

Memory Deallocation

- What happens when a method returns?
 - On the **runtime stack**:
 - The call frame is automatically popped off when the method returns
 - So, that memory is **deallocated**

1-23

Memory Deallocation

- What happens to **objects** on the heap?
 - An object stays on the heap even if there is no longer a variable referencing it!
 - So, Java has automatic **garbage collection**
 - It regularly identifies objects which no longer have a variable referencing them, and **deallocates** that memory

1-24