Objects and Arrays and Recursion, Oh my!

A look at what’s really going on

Where do variables and objects live?

Let’s lift the lid and peek inside the Java virtual machine ...

Stack

Heap

Code (binary)

```java
public static void main ( ... ) { int x;.. }
static void foo( ... ) { Muppet m; .... }
```

Stack

Heap

Code (binary)

```
main
x: 42
foo
m: -
bar
x: -y:
```

```
public static void main ( ... ) { int x;.. }
static void foo( ... ) { Muppet m; .... }
```
The Activation Stack

A sequence of “activation records” (a.k.a. stack frames)
One for each method invocation
Holds local variables
The actual contents of “primitive” types
(int, boolean, float, double, ...)
References to objects and arrays
(String, Muppet, int [], ...)

Recursion creates multiple activation records for a method

Stack

The stack grows and shrinks

```
foo ( ... ) {
    m: ...
    ...
    bar(12);
}

void bar(int x) {
    y: int = 13;
}
```

Consequences ...

When your program crashes, you get a “stack trace”:
Local variables and executing line from each current activation record

Very deep recursion can use up the stack
If you have an infinite recursive loop, this is how it will crash
Objects live in the heap

Strings are objects
Arrays are similar

The activation record (stack frame) holds a reference (pointer) to the object

static void foo( ... )
{ Muppet m;
  m = new Muppet(); }
A closer look at the object ...

Objects contain

A descriptor (reference to the class)
Fields (instance variables)

Arrays are Objects

Muppet [] ms;
ms = new Muppet[5];
ms[2] = new Muppet();

Arrays are Objects

Muppet [] ms;
ms = new Muppet[5];
ms[2] = new Muppet();
Arrays are Objects

Muppet [ ] ms;
ms = new Muppet[5];
ms[2] = new Muppet();

Heap

ms

muppet

Summary: Where things live

The stack contains an activation record for each active method execution
- Grows and shrinks
- Contains local variables, etc
The heap contains objects (also arrays, classes)
- Things with unknown sizes and lifetimes
Object references are pointers into the heap