Review of basic Java 3

**(contains some material from slides accompanying
Horstmann: Java for Everyone: Late Objects,
John Wiley and Sons Inc)**

Overview

- Review of basic Java:
  - Methods
    - Call and return
    - Parameters and results
    - Local variables
- With a focus on:
  - Formal syntax definition
  - Compiling schemes
- Later we will look at arrays:
  - Storage
  - Access
  - Algorithms

Methods as Black Boxes

- A method is a sequence of instructions with a name
  - You declare a method by defining a named block of code
  - You call a method in order to execute its instructions

```
public static void main(String[] args)
{
  double result = Math.pow(2, 3);
  . . .
}
```

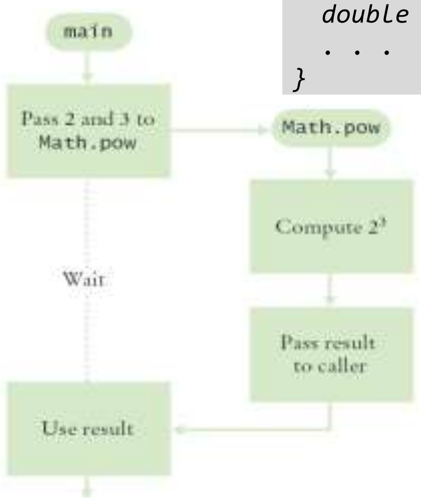
Method declaration

Method call

> *A method packages a computation consisting of multiple steps into a form that can be easily understood and reused*

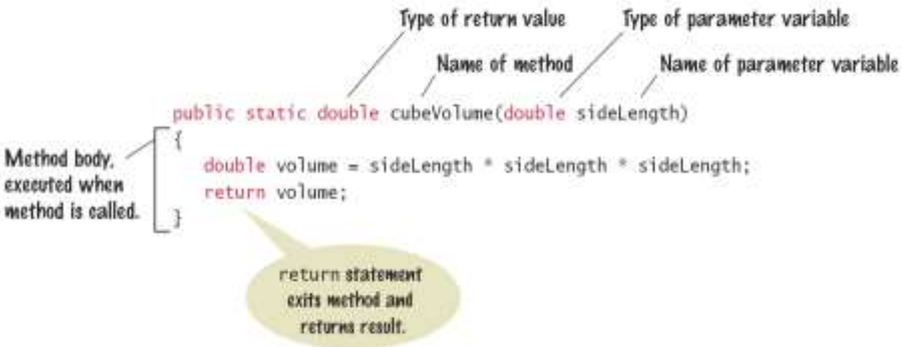
Flowchart of Calling a Method

```
public static void main(String[] args)
{
  double result = Math.pow(2, 3);
  . . .
}
```



- One method ‘calls’ another
  - main calls Math.pow()
  - Passes two arguments
    2 and 3
  - Math.pow starts
    Uses values (2, 3)
    Does its job
    Returns the answer
  - main uses result

Syntax: method declaration



```
public static double cubeVolume(double sideLength)
{
   double volume = sideLength * sideLength * sideLength;
   return volume;
}
```

Type of return value
Name of method
Type of parameter variable
Name of parameter variable
Method body, executed when method is called.
return statement exits method and returns result.

Syntax rules (slightly simplified)

MethodDeclaration: *MethodHeader MethodBody*

MethodHeader: *MethodModifiers* $_{opt}$ *Result MethodDeclarator*

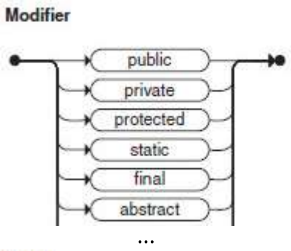
Result: *Type*
void

MethodDeclarator: *Identifier (FormalParameterList*$_{opt}$ *)*

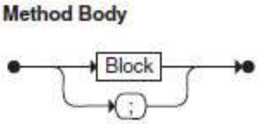
cont…

…continued as railroad diagrams:

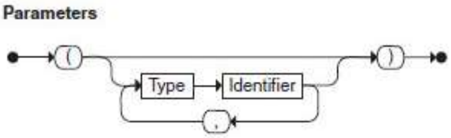
MethodModifier:



MethodBody:

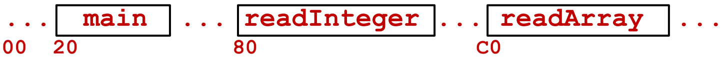


FormalParameterList:



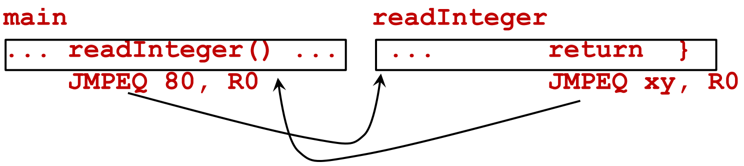
How are methods compiled?

- Each method is allocated its own, separate block of RAM
  - With start addresses set by the compiler
  - For example:



```
. . . [ main ] . . . [readInteger] . . . [readArray] . . .
00   20             80                   C0
```

- Control moves between the methods using unconditional jumps:
  - For example:



```
main                                readInteger
... readInteger() ...               ...         return   }
    JMPEQ 80, R0                                JMPEQ xy, R0
```

- But what should **`xy`** be if there are *several calls* of **`readInteger`**??

Coping with *return addresses*

- The return address must be the address of the instruction following the calling JMPEQ
  - *Wherever it was*
- A scheme for the Brookshear machine:
  - Before the calling JMPEQ: store the return address in a known location (attached to the called method)
  - Before the returning JMPEQ: overwrite **`xy`** with the stored return address
    Self-modifying code!
- Note:
  - This is (like) how the earliest programming languages worked
  - It does not work *in general* – in particular for *recursion*
  - More advanced CPUs have more powerful instructions

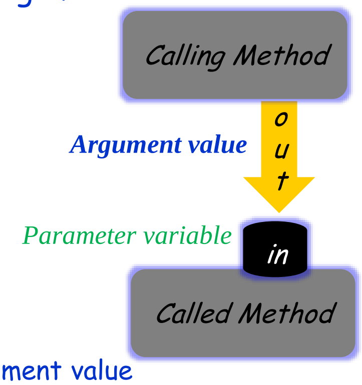
The details...

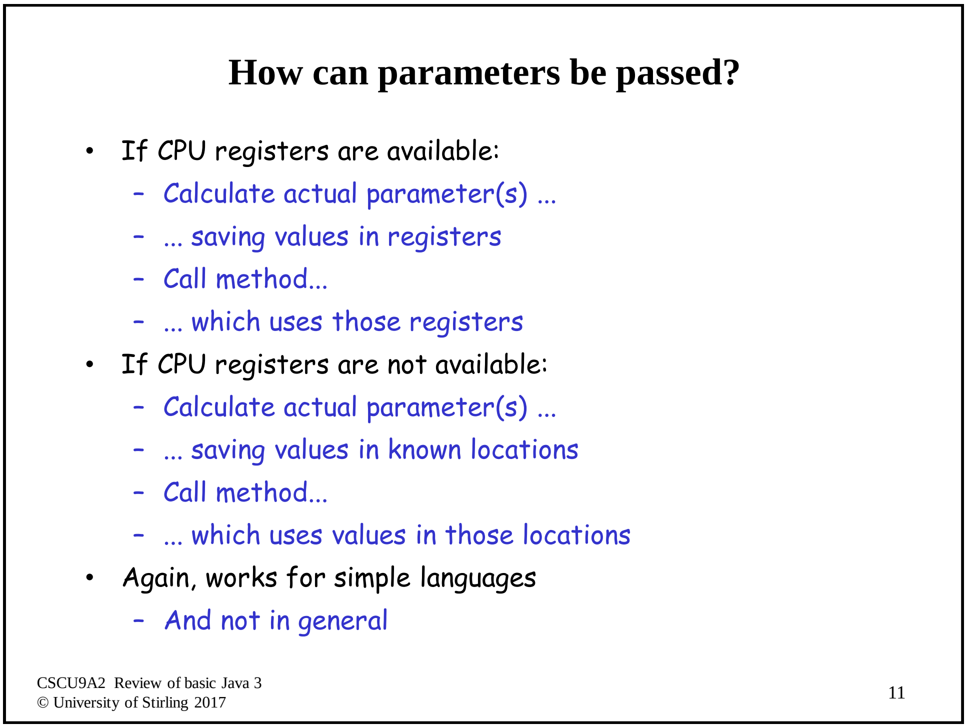
- Here is how `main` calling `readInteger` might compile:

```
        Addr   Instr
        20     ...                  Start of  main
        ...
      ⎡ 30     MOV 36 -> R0         Note return addr
Call ⎨  32     MOV R0 -> [7F]       Save in readInt's  known loc
      ⎣ 34     JMPEQ 80, R0         Jump to readInteger
        36     ...
        ...
        7F     00                   Reserved for return addr      ⎤
        80     ...                  Start of readInteger          ⎥
        ...                                                       ⎥ readInteger
        ⎡ 90   MOV [7F] -> R0       Retrieve return addr          ⎥
Return ⎨  92   MOV R0 -> [95]       Modify JMPEQ addr operand     ⎥
        ⎣ 94   JMPEQ 00, R0         Return jump                   ⎦
        ...
```

Parameter Passing

- Parameter variables receive the argument values supplied in the method call
  - They both must be the same type
- The argument value may be:
  - An expression
  - A 'literal' value
  - aka. *actual parameter* or argument
- The parameter variable is:
  - Declared in the called method
  - Initialized with the value of the argument value
  - Used as a variable inside the called method
  - aka. *formal parameter*



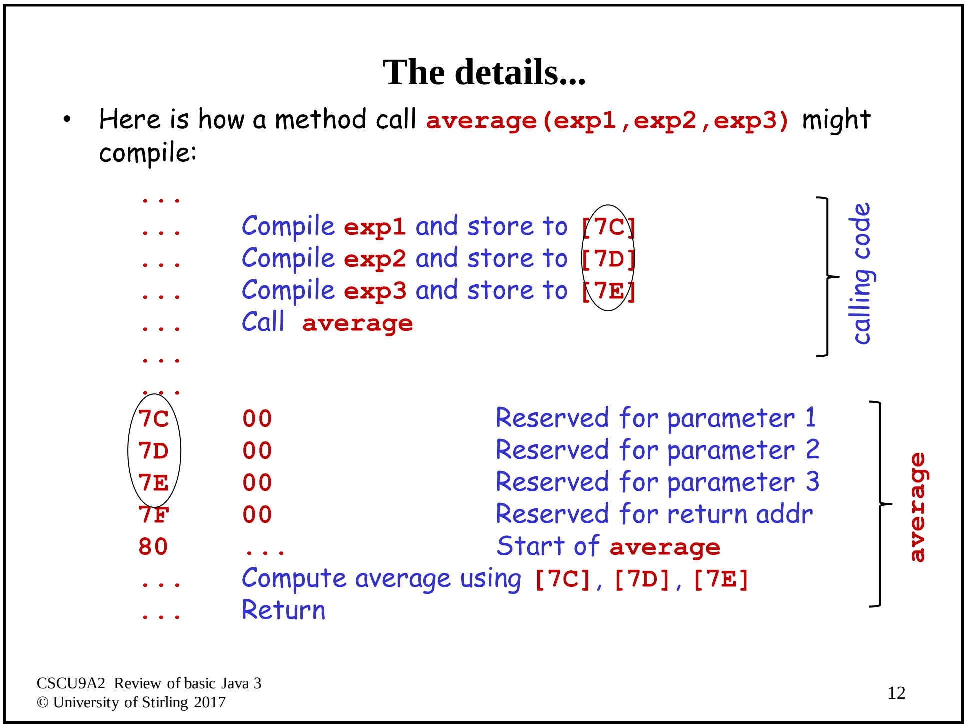


How can parameters be passed?

- If CPU registers are available:
  - Calculate actual parameter(s) ...
  - ... saving values in registers
  - Call method...
  - ... which uses those registers
- If CPU registers are not available:
  - Calculate actual parameter(s) ...
  - ... saving values in known locations
  - Call method...
  - ... which uses values in those locations
- Again, works for simple languages
  - And not in general

CSCU9A2 Review of basic Java 3
© University of Stirling 2017

11



The details...

- Here is how a method call **average(exp1,exp2,exp3)** might compile:

```
...
...    Compile exp1 and store to [7C]
...    Compile exp2 and store to [7D]
...    Compile exp3 and store to [7E]
...    Call  average
...
...
7C     00                Reserved for parameter 1
7D     00                Reserved for parameter 2
7E     00                Reserved for parameter 3
7F     00                Reserved for return addr
80     ...               Start of average
...    Compute average using [7C], [7D], [7E]
...    Return
```

calling code
average

CSCU9A2 Review of basic Java 3
© University of Stirling 2017

12

Return Values

- Methods can (optionally) return one value
  - Declare a return type in the method declaration
  - Add a return statement that returns a value
  - A return statement does two things:
    1) Immediately terminates the method
    2) Passes the return value back to the calling method

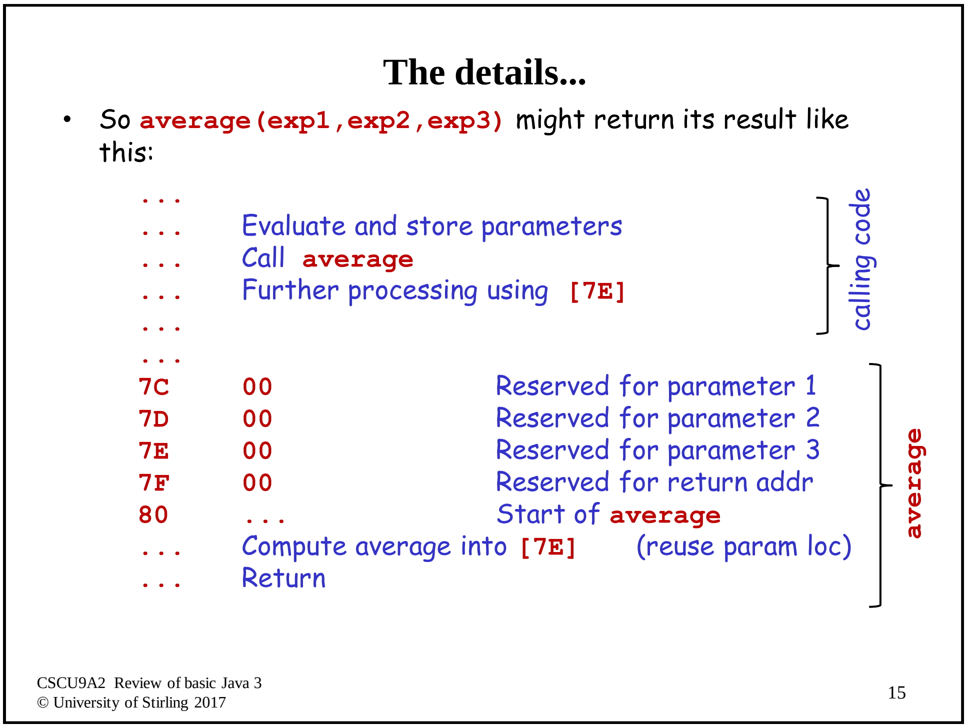
return type

```
public static double cubeVolume (double sideLength)
{
  double volume = sideLength * sideLength * sideLength;
  return volume;
}
```

return statement

How results be returned?

- If there is no `return expr`: nothing to be done
- To return a result: same scheme as parameters:
  - Compute value of expression `expr`
  - Put in a CPU register if available…
  - … or a known location – could even re-use a parameter location
  - Return
  - Calling code accesses the register or known location
- Again, works for simple languages
  - And not in general



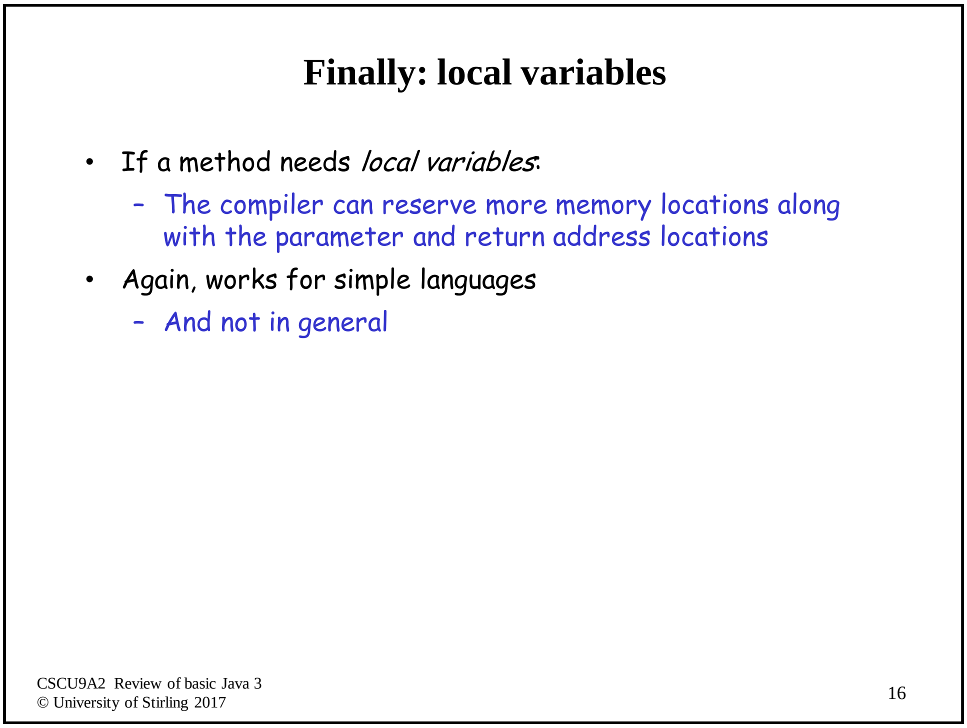
The details...

- So **average(exp1,exp2,exp3)** might return its result like this:

```
...
...     Evaluate and store parameters
...     Call  average
...     Further processing using  [7E]
...
...
7C      00                      Reserved for parameter 1
7D      00                      Reserved for parameter 2
7E      00                      Reserved for parameter 3
7F      00                      Reserved for return addr
80      ...                     Start of average
...     Compute average into [7E]       (reuse param loc)
...     Return
```

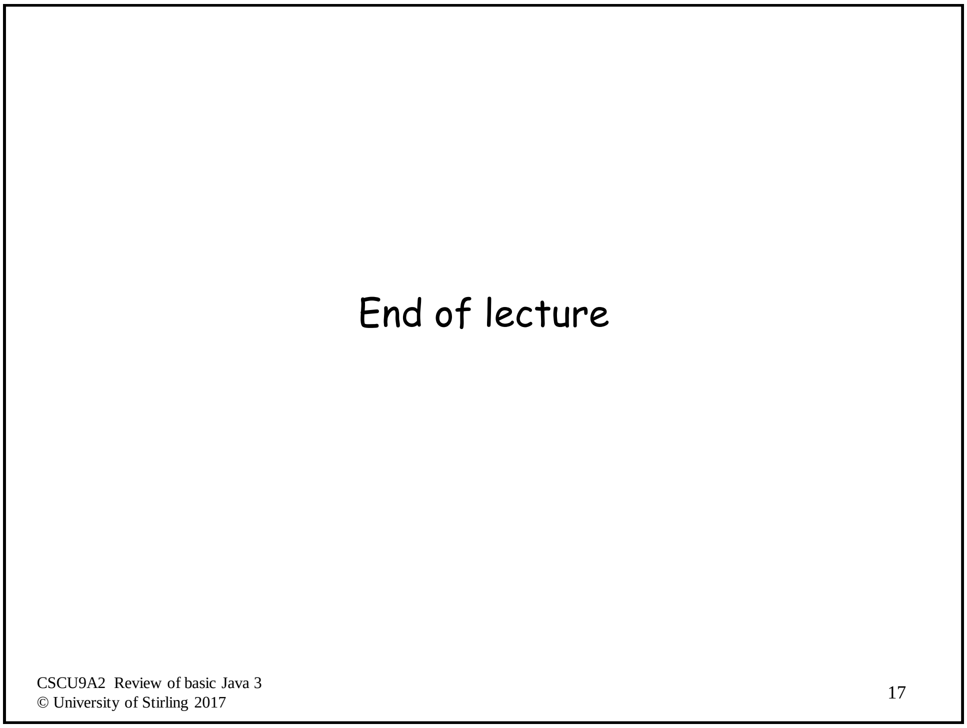
calling code

average



Finally: local variables

- If a method needs *local variables*:
  - The compiler can reserve more memory locations along with the parameter and return address locations
- Again, works for simple languages
  - And not in general



End of lecture