Methods and Memory

- Whenever a method is invoked, memory is needed on the runtime stack.
- Methods need memory for:
  - Arguments
  - Local variables
  - Administrative values (to keep track of program flow)

The memory associated with a single method is a stack frame.
- A stack frame exists as long as the owning method is active.
- When the method is no longer active, the stack frame is no longer needed.

---

Stack Frame Example

```java
1. int h, i;
2. 3. void A(int x, int y) {
4.   boolean i, j;
5.   B(h);
6.   // more code...
7. }
8. 9. void B(int w) {
10.  int j, k;
11.  i = 2 * w;
12.  w = w + 1;
13.  // more code...
14. }
15. 16. void main() {
17.     h = 5; a = 3; b = 2;
18.     A(a, b)
19. }
```

On program activation, `main()` starts with all variables undefined, and a single static link back to the static area of memory.

---

Stack Frame Example

```java
1. int h, i;
2. 3. void A(int x, int y) {
4.   boolean i, j;
5.   B(h);
6.   // more code...
7. }
8. 9. void B(int w) {
10.  int j, k;
11.  i = 2 * w;
12.  w = w + 1;
13.  // more code...
14. }
15. 16. void main() {
17.     h = 5; a = 3; b = 2;
18.     A(a, b)
19. }
```

As program runs, `main()` changes values both in its local stack frame and in the static memory area.
Stack Frame Example

1. int h, i;
2. void A(int x, int y) {
3.     boolean i, j;
4.     B(h);
5.     // more code...
6. }
7. }
8. void B(int w) {
9.     int j, k;
10.    i = 2 * w;
11.    w = w + 1;
12.    // more code...
13. }
14. }
15. void main() {
16.    int a, b;
17.    h = 5; a = 3; b = 2;
18.    A(a, b)
19. }

Review: A More Complex Language

Program → Declaration* Function* Main
Function → Void | NonVoid
    Void → void Name(Parameters) { Statement* }
NonVoid → int Name(Parameters) { Statement* Return}
Name → Letter Letter*
Parameters → ε | int Variable(, int Variable)*
Return → return Expression;
Main → main() { Statement* }
Statement → Declaration | Assignment | Call
Declaration → int Variable; | int Assignment
Assignment → Variable = Expression;
Expression → Int | Variable | BinaryOperation | Call
Call → Name(Arguments);
Arguments → ε | Argument(, Argument)*
Argument → Variable | Int

Rest of the language (Variable, Int, etc.) is the same as it was before

Stack Frame Example

1. int h, i;
2. void A(int x, int y) {
3.     boolean i, j;
4.     B(h);
5.     // more code...
6. }
7. }
8. void B(int w) {
9.     int j, k;
10.    i = 2 * w;
11.    w = w + 1;
12.    // more code...
13. }
14. }
15. void main() {
16.    int a, b;
17.    h = 5; a = 3; b = 2;
18.    A(a, b)
19. }

A Program in the Function Language

1. int x = 0;
2. int y = 0;
3. void addToXY(int i, int j) {
4.     x = x + i;
5.     y = y + j;
6. }
7. }
8. int sumXY() {
9.     return x + y;
10. }
11. }
12. main() {
13.     int a = 1;
14.     int b = 2;
15.     addToXY(a, b);
16.     int c = sumXY();
17.     A(a, b)
18. }

What would a parse (derivation) tree for this program look like, exactly?
Review: Rules for Functions and Calls

- Our syntax guarantees that every non-void function has a return statement, and that no void function does.
- Other features must be explicitly written as semantic rules:
  1. The expression following the return statement in a non-void function must have an integer value.
  2. Every call to any function must have the same number of arguments as the function has parameters.
  3. If a function is called in an expression on the right-hand side of an assignment statement, it must be a non-void function.
  4. A function call in a statement that is not an assignment can be either void or non-void.

Note: since everything is of integer type in this language, we don’t have to specify a rule like #2 for parameter types, as in more complex languages.

This Week

- Topic: Semantics of functions/methods
- Reading: Text, 9.1–9.3
- Meet: usual schedule
- Homework 03: due Friday, 12 April, 5:00 PM
- Office Hours: Wing 210
  - Monday, 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Wednesday: 9:00 AM – 10:30 AM
  - Thursday, 2:00 PM – 3:00 PM
  - Friday: 9:00 AM – 10:30 AM