

Lecture 13

# **Call Stacks & Debugging**

# Announcements for This Lecture

---

## Prelim 1

---

- TONIGHT 7:30-9pm
  - Abel–Price (Upson B17)
  - Rabbit–Teo (Upson 111)
  - Ting–Zytariuk (Upson 109)
- Graded late tonight
  - Will have grade Fri morn
  - In time for drop day
- Make-up, Friday 4:30
  - For preapproved students

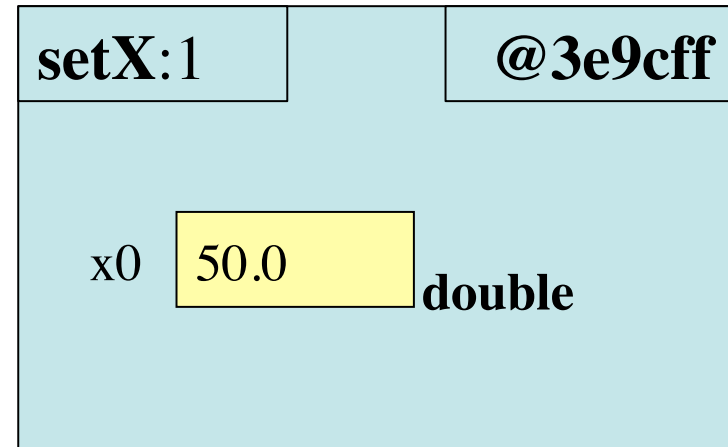
## Assignments

---

- A4 due next Thursday
  - Last weekend to work on it
  - Should be straight-forward after this exam.
  - Graded when you get back
- A5 posted next Thursday
  - Have 1.5 weeks after Spring Break to do it
  - Welcome, but not expected, to do it over the break

# Review: Method Calls

- Method calls require frames
  - Model how the call works
- Steps to the method call:
  1. Draw a frame for the call
  2. Assign the argument value to the parameter (in frame)
  3. Execute the method body
    - Look for variables in the frame
    - If not there, look in folder given by the scope box
  4. Erase the frame for the call



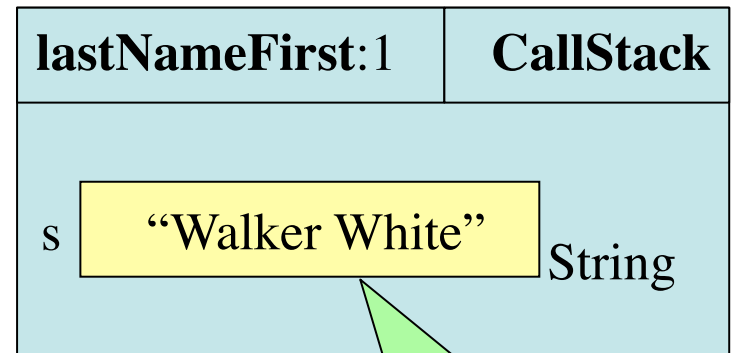
```
public void setX(double x0) {  
    x = x0;  
}
```

Only at the End!

# Frames and Helper Methods

```
/** Precondition: s is in form
 * <first-name> <last-name> */
public static String
    lastNameFirst(String s) {
    String first = firstName(s); // Line 1
    String last = lastName(s); // Line 2
    return last+","+first;      // Line 3
}
```

```
/** Precondition: see lastNameFirst */
public static String firstName(String s) {
    int end = s.indexOf(" ")
    return s.substring(0,end);
}
```

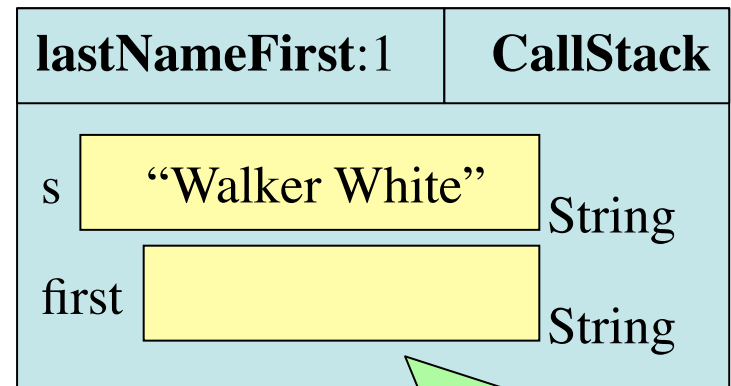
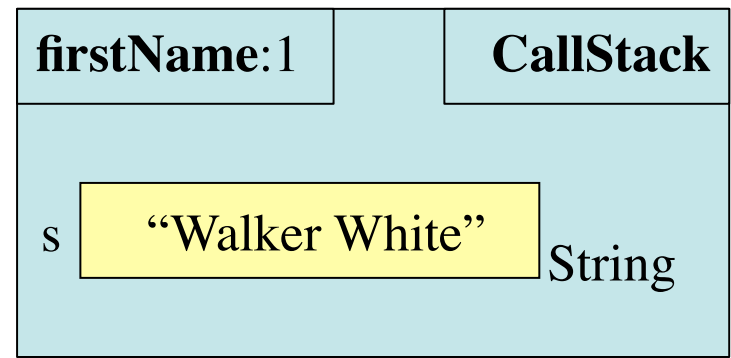


String an object.  
This is "shorthand"

# Frames and Helper Methods

```
/** Precondition: s is in form
 * <first-name> <last-name> */
public static String
    lastNameFirst(String s) {
    String first = firstName(s); // Line 1
    String last = lastName(s); // Line 2
    return last+", "+first; // Line 3
}
```

```
/** Precondition: see lastNameFirst */
public static String firstName(String s) {
    int end = s.indexOf(" ")
    return s.substring(0,end);
}
```



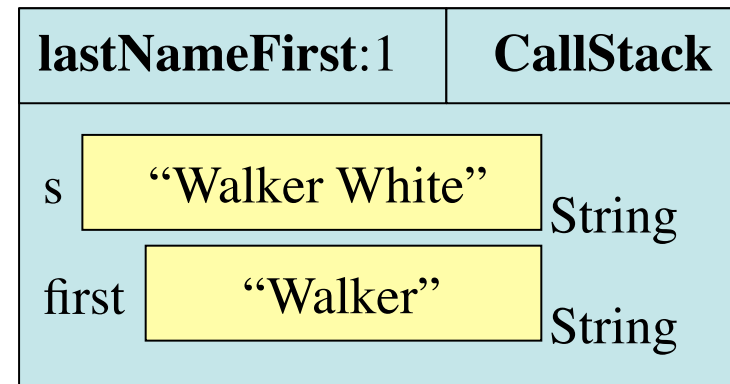
Not done. Do not erase!

# Frames and Helper Methods

```
/** Precondition: s is in form
 * <first-name> <last-name> */
public static String
    lastNameFirst(String s) {
    String first = firstName(s); // Line 1
    String last = lastName(s); // Line 2
    return last+", "+first; // Line 3
}
```

```
/** Precondition: see lastNameFirst */
public static String firstName(String s) {
    int end = s.indexOf(" ");
    return s.substring(0,end);
}
```

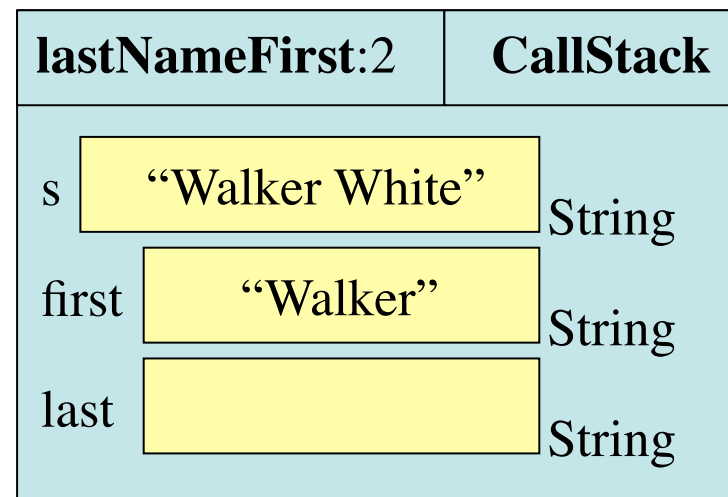
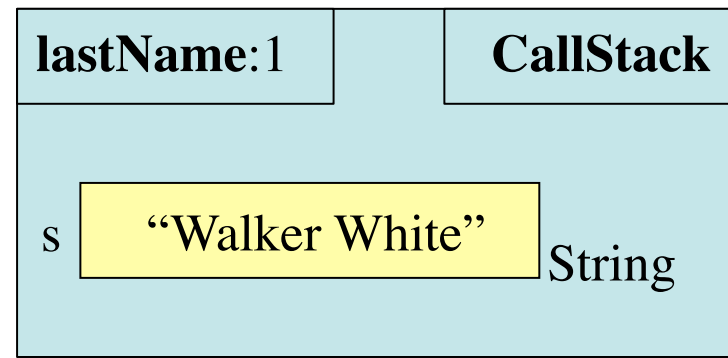
*ERASE WHOLE FRAME*



# Frames and Helper Methods

```
/** Precondition: s is in form
 * <first-name> <last-name> */
public static String
    lastNameFirst(String s) {
    String first = firstName(s); // Line 1
    String last = lastName(s); // Line 2
    return last+", "+first;      // Line 3
}
```

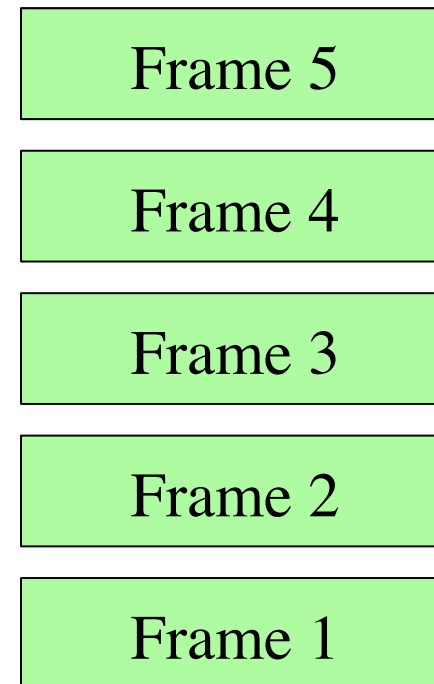
```
/** Precondition: see lastNameFirst */
public static String firstName(String s) {
    int end = s.indexOf(" ");
    return s.substring(0,end);
}
```



# The Call Stack

---

- Methods are “stacked”
  - Like a stack of plates
  - Cannot remove one below w/o removing above
- Stack represents memory “high water mark”
  - Must have enough to keep the stack in memory
  - StackOverflow if cannot hold the entire stack.





# Review: Fibonacci Sequence

---

- Sequence of numbers:  
1, 1, 2, 3, 5, 8, 13, ...

- Recursive definition:

- **Recursive Case**

$$a_n = a_{n-1} + a_{n-2}$$

- **Base Case**

$$a_0 = 1$$

- **(another) Base Case**

$$a_1 = 1$$

```
/** Yields: Fibonacci number  $a_n$ 
```

```
* Precondition:  $n \geq 0$  */
```

```
public static int fib(int n) {
```

```
    if (n <= 1) {
```

```
        return 1;
```

```
    }
```

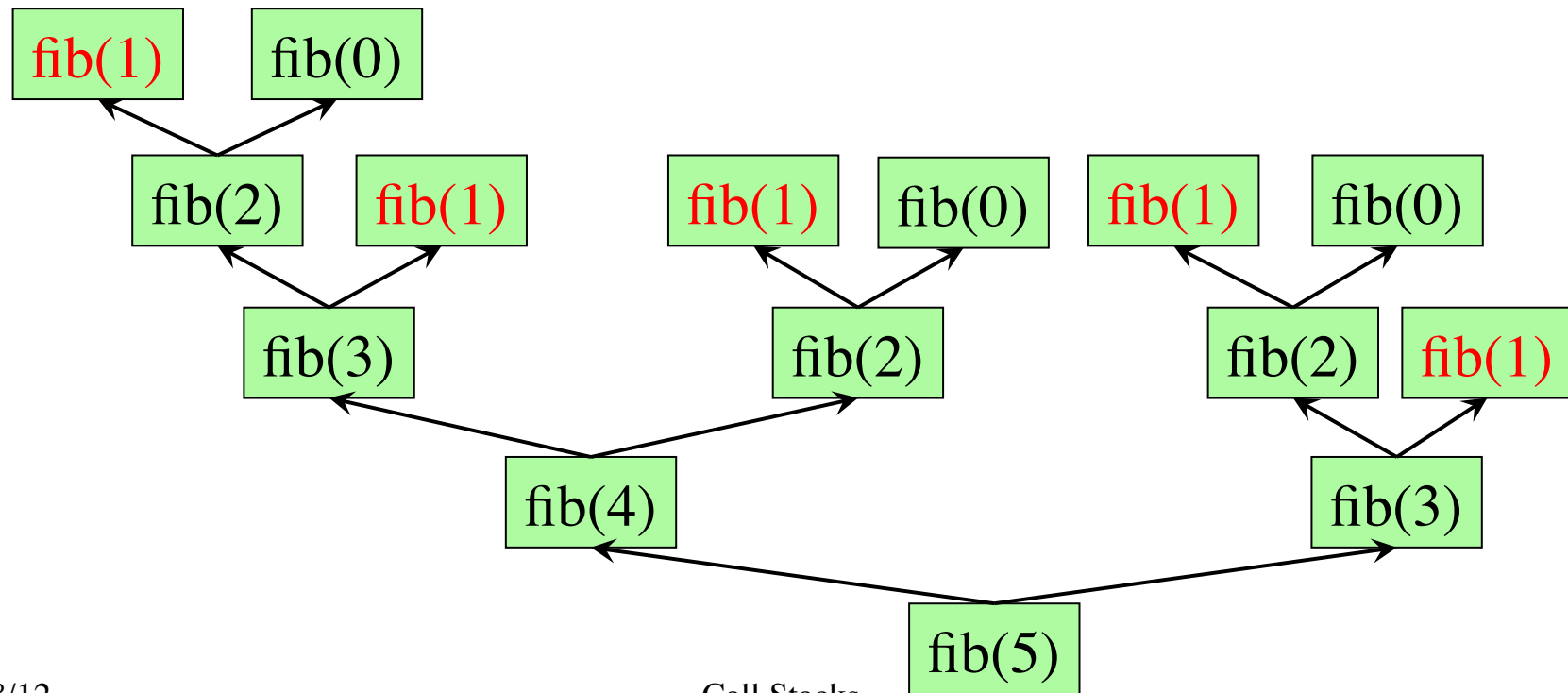
```
    return fib(n-1)+
```

```
        fib(n-2);
```

```
}
```

# Fibonacci: # of Frames vs. # of Calls

- Fibonacci is very inefficient.
  - $\text{fib}(n)$  has a stack that is always  $\leq n$
  - But  $\text{fib}(n)$  makes a lot of **redundant calls**



# Debugging and the Call Stack

- Errors or **Exceptions** in Java give current call stack

```
06 /** Yields: String s truncated .... */
07 public static String truncate5(String s) {
08     int b = 10 / 0;
09     if (s.length() <= 5) { return s; }
10
11     return s.substring(0,5);
12 }
```

Turn on line numbering  
in DrJava. Preferences /  
Display Options

important part

ArithmeticException: / by zero

at A4Methods.truncate5(A4Methods.java:8)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(...java:39)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(...java:25)

at java.lang.reflect.Method.invoke(Method.java:585)

call stack

# Debugging Strategies

---

## Black Box Testing

---

- Method is “opaque”
  - Test looks at what it does
  - Functions: what it returns
  - Procedures: what changes
- **Example:** JUnit tests
- **Problems:**
  - Are the tests everything?
  - What caused the error?

## White Box Testing

---

- Method is “transparent”
  - Tests/debugging takes place inside of the method
  - Focuses on where error is
- **Example:** ??????
- **Problems:**
  - Much harder to do
  - Hurts code performance

# Assert Statements

---

`assert <boolean>; // Creates Exception if <boolean> false`

`assert <boolean> : <String>; // As above, but displays <String>`

- Can write and forget
  - Only used if debugging turned on in Java
  - Otherwise, Java treats it like a comment
- Code defensively!

```
/** Set worker's last name to n
 * Precondition: Cannot be null */
public void setName(String n) {
    assert n != null;
    lname = n;
}
```

# Print-Statements: Invasive Debugging

---

- `System.out.print(String)`:
  - Prints a String to console (next to Interaction Pane)
  - Works outside of DrJava (if no Interactions Pane)
- `System.out.println(String)`:
  - Same, but provides a “carriage return”
- **Strategy**: Put inside method to see what it does
  - **Variables**: look at current value
  - **Traces**: figure out the “code path”

# Tracing with Print Statements

---

- Method to right is buggy
  - `w.setName("White")` sets the name to null (???)
  - Violates the specification
- Observation: not all code should be executed
  - Should skip over `if`
  - Use traces to see if it actually skips over `if`

```
public class Worker {  
    ...  
    /** Sets name to n; "<none>"  
     * if n null or "" */  
    public void setName(String n) {  
        name = n;  
        if (n == null || n.equals("")); {  
            name = "<none>";  
        }  
    }  
    ...  
}
```

# Tracing with Print Statements

## Console

- 1: ...
- 2: White
- 3: White
- 4: <none>

After a command

At the start

```
public void setName(String n) {  
    System.out.println("1:" + name);  
    name = n;  
    System.out.println("2:" + name);  
    if (n == null || n.equals("")); {  
        System.out.println("3:" + name);  
        name = "<none>";  
        System.out.println("4:" + name);  
    }  
}
```

### Invasive

Must remove all statements from “shipping” code.



# What is the Error?

```
public void setName(String n) {  
    System.out.println("1:" + name);  
    name = n;  
    System.out.println("2:" + name);  
    if (n == null || n.equals("")) {  
        System.out.println("3:" + name);  
        name = "<none>";  
        System.out.println("4:" + name);  
    }  
}
```

Here!

- if-statements do the command after the )
  - If {}, does all of the commands inside {}
  - If ; it does nothing