Stack Use

What is a stack?

- A stack is a Last In, First Out (LIFO) data structure
- Anything added to the stack goes on the “top” of the stack
- Anything removed from the stack is taken from the “top” of the stack
- Things are removed in the reverse order from that in which they were inserted

Constructing a stack

- To use Java's built-in stacks, you need either of
  - `import java.util.*;`
  - `import java.util.Stack;`

- There is just one stack constructor:

```
Stack<E> stack = new Stack<E>();
```

- `E` is the type of element (for example, `String`) that you intend to put on the stack
  - To get the old (pre-Java 5) behavior, where objects of any kind can be put on the stack, use **`<?>`**

Fundamental stack operations

`oldItem = stack.push(item)`

- Adds the item (of type `E`) to the top of the stack; the item pushed is also returned as the value of `push`

`item = stack.pop();`

- Removes the `E` at the top of the stack and returns it

`item = stack.peek();`

- Returns the top `E` of the stack but does not remove it from the stack

`stack.empty()`

- Returns `true` if there is nothing in the stack

Additional stack operation

```
int i = stack.search(item);
```

- Returns the *1-based* position of the element on the stack. That is, the top element is at position 1, the next element is at position 2, and so on.
- Returns -1 if the element is not on the stack
- This is not a “conventional” stack operation, but is sometimes useful

Inheritance vs. composition

- This is inheritance: `class X extends class Y { ... }`
  - X inherits all the (non-private) methods and variables of Y
  - This is appropriate if X *is a kind of* Y, but not otherwise
    - Because you get *all* of Y, whether it's appropriate for X or not
  - Inheritance is often overused
- This is composition: `class X { Y myYvariable; ... }`
  - To make a method, say `int m(int a)`, of class Y available to objects of class X, you use delegation:
    `class X { ...; int m(int a) { return myYVariable.m(a); } ... }`
  - Similarly, class X can have getters and setters that refer to variables of the object `myYVariable`
  - Composition is appropriate if class X *uses* class Y, but isn't *a kind of* class Y
- If in doubt, use composition rather than inheritance

Stack ancestry

- The Stack class extends (inherits from) the Vector class
  - Hence, anything you can do with a Vector, you can also do with a Stack
    - For example, you can index into it, add elements to and remove elements from the middle, etc.
  - This is not how stacks should be used!
    - If you want Vector operations, use a Vector--the Stack methods give you conventional names for stack operations, but no new capabilities
    - A “stack” is a very specific data structure, defined by the preceding operations; it just happens to be *implemented* by extending Vector
    - Even Sun gets things wrong sometimes
- The Vector class implements the Collection interface
  - Hence, anything you can do with a Collection, you can also do with a Stack
  - The most useful operation this gives you is toArray()

Some uses of stacks

- Stacks are used for:
  - Any sort of nesting (such as parentheses)
  - Evaluating arithmetic expressions (and other sorts of expression)
  - Implementing function or method calls
  - Keeping track of previous choices (as in backtracking)
  - Keeping track of choices yet to be made (as in creating a maze)

A balancing act

- ([]({()}[()])) is balanced; ([]({()}[())]) is not
- Simple counting is not enough to check balance
- You can do it with a stack: going left to right,
  - If you see a (, [, or {, push it on the stack
  - If you see a),], or }, pop the stack and check whether you got the corresponding (, [, or {
  - When you reach the end, check that the stack is empty

Expression evaluation

- Almost all higher-level languages let you evaluate expressions, such as `3*x+y` or `m=m+1`
- The simplest case of an expression is one number (such as `3`) or one variable name (such as `x`)
  - These *are* expressions
- In many languages, `=` is considered to be an operator
  - Its value is (typically) the value of the left-hand side, after the assignment has occurred
- Situations sometimes arise where you want to evaluate expressions yourself, without benefit of a compiler

Performing calculations

- To evaluate an expression, such as 1+2*3+4, you need *two* stacks: one for operands (numbers), the other for operators: going left to right,
  - If you see a number, push it on the number stack
  - If you see an operator,
    - **While** the top of the operator stack holds an operator of equal or higher precedence:
      - pop the old operator
      - pop the top two values from the number stack and apply the old operator to them
      - push the result on the number stack
    - push the new operator on the operator stack
  - At the end, perform any remaining operations

Example: 1+2*3+4

- 1 : push 1 on number stack
- + : push + on op stack
- 2 : push 2 on number stack
- * : because * has higher precedence than +, push * onto op stack
- 3 : push 3 onto number stack
- + : because + has lower precedence than *:
  - pop 3, 2, and *
  - compute 2*3=6, and push 6 onto number stack
  - push + onto op stack
- 4 : push 4 onto number stack
- end : pop 4, 6 and +, compute 6+4=10, push 10; pop 10, 1, and +, compute 1+10=11, push 11
- 11 (at the top of the stack) is the answer

Handling parentheses

- When you see a left parenthesis, **(**, treat it as a low-priority operator, and just put it on the operator stack
- When you see a right parenthesis , **)**, perform all the operations on the operator stack until you reach the corresponding left parenthesis; then remove the left parenthesis

Handling variables

- There are two ways to handle variables in an expression:
  - When you encounter the variable, look up its value, and put its value on the operand (number) stack
    - This simplifies working with the stack, since everything on it is a number
  - When you encounter a variable, put the variable itself on the stack; only look up its value later, when you need it
    - This allows you to have embedded assignments, such as `12 + (x = 5) * x`

Handling the = operator

- The assignment operator is just another operator
  - It has a lower precedence than the arithmetic operators
  - It should have a higher precedence than **(**
- To evaluate the **=** operator:
  - Evaluate the right-hand side (this will already have been done, if **=** has a low precedence)
  - Store the value of the right-hand side into the variable on the left-hand side
    - You can only do this if your stack contains variables as well as numbers
  - Push the value onto the stack

At the end

- Two things result in multiple special cases
  - You frequently need to compare the priority of the current operator with the priority of the operator at the top of the stack—but the stack may be empty
  - Earlier, I said: “At the end, perform any remaining operations”
- There is a simple way to avoid these special cases
  - Invent a new “operator,” say, `$`, and push it on the stack initially
  - Give this operator the lowest possible priority
  - To “apply” this operator, just quit—you’re done

Some things that can go wrong

- The expression may be ill-formed:

  `2 + 3 +`

  - When you go to evaluate the second `+`, there won't be two numbers on the stack

  `1 2 + 3`

  - When you are done evaluating the expression, you have more than one number on the stack

  `(2 + 3`

  - You have an unmatched `(` on the stack

  `2 + 3)`

  - You can't find a matching `(` on the stack
- The expression may use a variable that has not been assigned a value

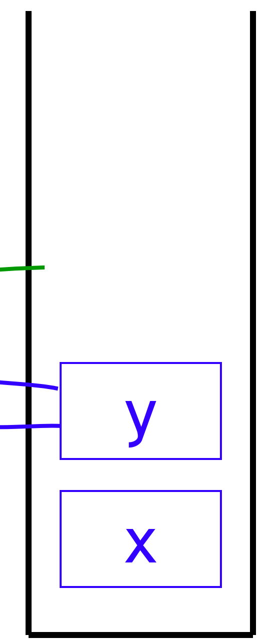
Types of storage

- In almost all languages (including Java), data is stored in two different ways:
  - Temporary variables—parameters and local variables of a method—are stored in a *stack*
    - These values are popped off the stack when the method returns
    - The value returned from a method is also temporary, and is put on the stack when the method returns, and removed again by the calling program
  - More permanent variables—objects and their instance variables and class variables—are kept in a *heap*
    - They remain on the heap until they are "freed" by the programmer (C, C++) or garbage collected (Java)

Stacks in Java

- Stacks are used for local variables (including parameters)

```
void methodA() {
    int x, y; // puts x, y on stack
    y = 0;
    methodB();
    y++;
}
void methodB() {
    int y, z; // puts y, z on stack
    y = 5;
    return; // removes y, z
}
```



Recursion

- A recursive definition is when something is defined *partly* in terms of itself
- Here's the mathematical definition of factorial:

$$\text{factorial}(n) = \begin{cases} 1, \text{ if } n <= 1 \\ n * \text{factorial}(n - 1) \text{ otherwise} \end{cases}$$

- Here's the programming definition of factorial:

```
static int factorial(int n) {
    if (n <= 1) return 1;
    else return n * factorial(n - 1);
}
```

Supporting recursion

```
static int factorial(int n) {
    if (n <= 1) return 1;
    else return n * factorial(n - 1);
}
```

- If you call x = factorial(3), this enters the factorial method with n=3 on the stack
- | factorial calls itself, putting n=2 on the stack
- | | factorial calls itself, putting n=1 on the stack
- | | factorial returns 1
- | factorial has n=2, computes and returns 2*1 = 2
- factorial has n=3, computes and returns 3*2 = 6

Factorial (animation 1)

- `x = factorial(3)`

3 is put on stack as n

- ```java
  static int factorial(int n) { //n=3
      int r = 1;
      if (n <= 1) return r;
      else {
          r = n * factorial(n - 1);
          return r;
      }
  }
  ```

r is put on stack with value 1

All references to r use this r

All references to n use this n

Now we recur with 2...

r=1

n=3
  ```

Factorial (animation 2)

- ```
  r = n * factorial(n - 1);
  ```

2 is put on stack as n

- ```
  static int factorial(int n) {//n=2
      int r = 1;
      if (n <= 1) return r;
      else {
          r = n * factorial(n - 1);
          return r;
      }
  }
  ```

r is put on stack with value 1

Now using this r

And this n

r=1

n=2

r=1

n=3

Now we recur with 1...

Factorial (animation 3)

- `r = n * factorial(n - 1);`

1 is put on stack as n

- ```
  static int factorial(int n) {
      int r = 1;
      if (n <= 1) return r;
      else {
          r = n * factorial(n - 1);
          return r;
      }
  }
  ```

Now using this r

r is put on stack with value 1

And this n

Now we pop r and n off the stack and return 1 as factorial(1)

r=1

n=1

r=1

n=2

r=1

n=3
  ```

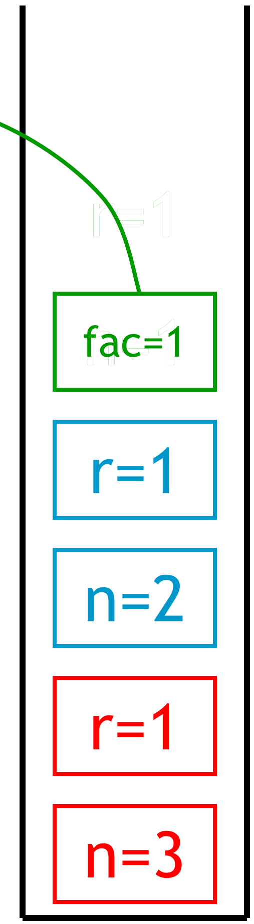
Factorial (animation 4)

- `r = n * factorial(n - 1);`

- ```
  static int factorial(int n) {
      int r = 1;
      if (n <= 1) return r;
      else {
          r = n * factorial(n - 1);
          return r;
      }
  }
  ```

Now using this r

And this n



Now we pop r and n off the stack and return 1 as factorial(1)
  ```

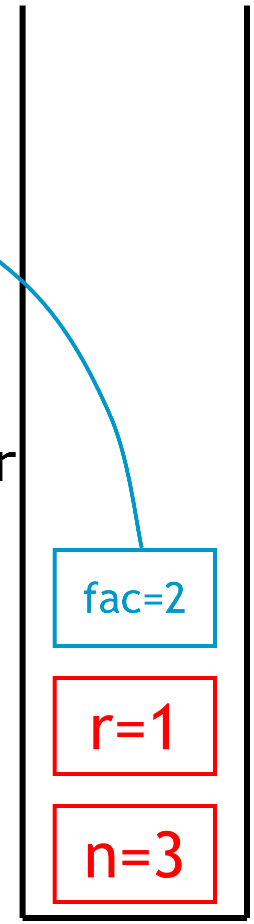
Factorial (animation 5)

- `r = n * factorial(n - 1);`

- ```
  static int factorial(int n) {
      int r = 1;
      if (n <= 1) return r;
      else {
          r = n * factorial(n - 1);
          return r;
      }
  }
  ```

Now using this r

And
this n



1

2 * 1 is 2;

Pop r and n;

Return 2
  ```

Factorial (animation 6)

- `x = factorial(3)`

-
```
static int factorial(int n) {
    int r = 1;
    if (n <= 1) return r;
    else {
        r = n * factorial(n - 1);
        return r;
    }
}
```

2

3 * 2 is 6;

Pop r and n;

Return 6

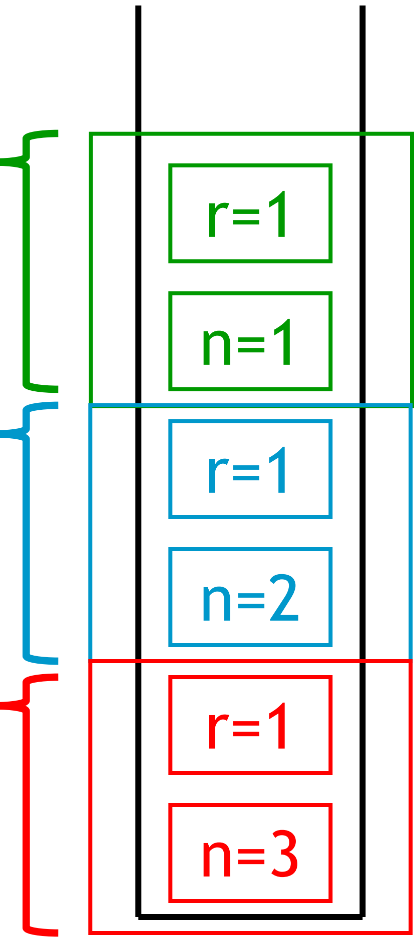
Now using this r

And this n

fac= 6

Stack frames

- Rather than pop variables off the stack one at a time, they are usually organized into stack frames
- Each frame provides a set of variables and their values
- This allows variables to be popped off all at once
- There are several different ways stack frames can be implemented



Summary

- Stacks are useful for working with any nested structure, such as:
  - Arithmetic expressions
  - Nested statements in a programming language
  - ***Any*** sort of nested data

The End

http://www.gottatoy.com/Fisher-Price-Brilliant-Basics-Rock-A-Stack-Baby-Toy_p_856.html