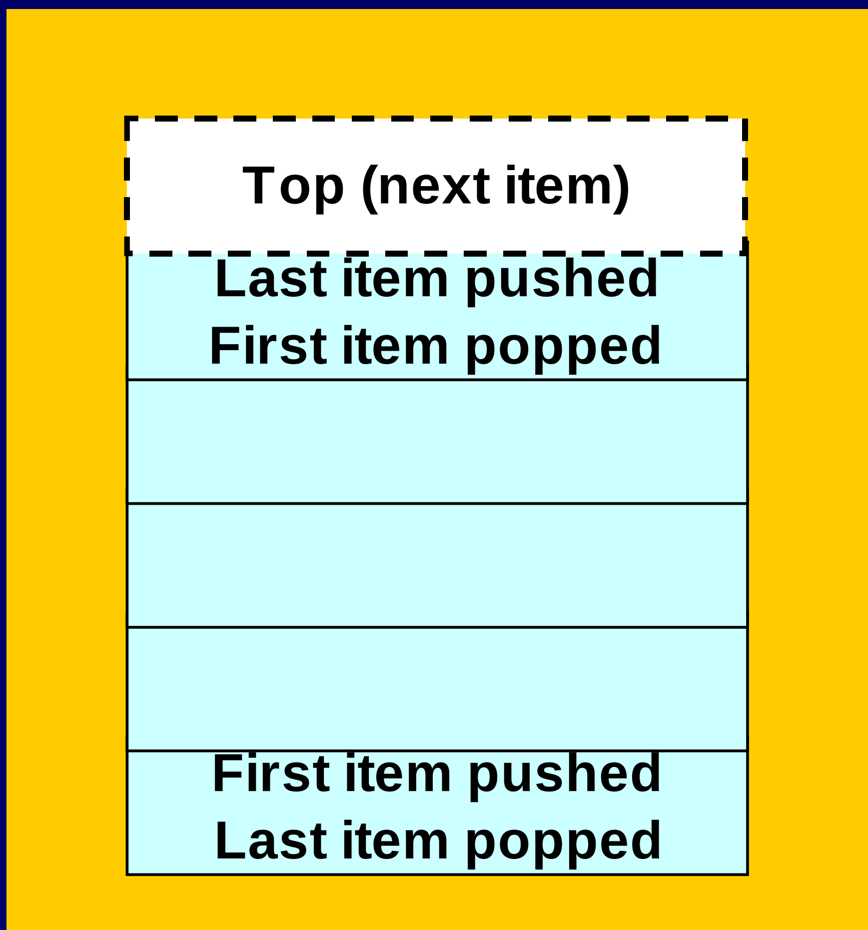
Monday, July 6

**First exam**

# Stacks




- LIFO data structure
  - Last In, First Out
- All items except last item pushed are inaccessible
- So has very few possible operations:
  - `push`, `pop`, `peek`, `isEmpty`, `isFull`, `size`, `clear`
- Lots of applications

# Applying stacks

- Can be used to eliminate recursion
  - Iteration and stacks instead of recursive calls
    - For each "recursive" step
      - Push critical data values
    - While stack is not empty
      - Pop values – like "return" from recursive call
  - It's how the compiler does it
    - Pushes "activation record" (a.k.a., "stack frame") for every function call, not just recursive ones
- In fact, idea applies to *any nested structure*
  - Recursion is just a nesting of function calls
  - What about nested parentheses in expressions?

# Stack interface for general data

- Store `Object` data items (or `<T>`)

```
void push(Object item); // push item on stack
Object pop(); // pop top item from stack
```

  - So can refer to anything – even other stacks!
    - No need to reprogram stack for every application
- User works a little harder to use though
  - Easiest to do with utility methods like:

```
void pushInt(int value, Stack stack);
    // creates Integer object and pushes it on the stack
int popInt(Stack stack);
    // pops from stack, casts, and gets int value from object
```

# Checking balanced ( ), [ ], { }

- Okay to nest, like `{x/[y*(a+b)]}`
- Not okay to mismatch (or nest improperly)
  - `(a/(x + y)` is missing a right parenthesis
  - `( x + [y-2)]` is mismatched at `[ )`
- Parentheses fully match if the following works:

```
for (each character in the expression) {
    if a left parenthesis - push it on the stack;
    if a right parenthesis
            pop matching left parenthesis from stack
} stack is empty at the end
```

# Postfix (and prefix) notation

- Also called "reverse Polish" – reversed form of notation devised by mathematician named Jan Łukasiewicz (so really lü-kä-sha-vech notation)
- Infix notation is: `operand operator operand`
  - Like `4 + 22`
  - Requires parentheses sometimes: `5 * (2 + 19)`
- Postfix form is: `operand operand operator`
  - So `4 22 +`
  - No parentheses required: `5 2 19 + *`
- Prefix is `operator operand operand`: `+ 4 22`

# Evaluating postfix expressions

- Algorithm (start with an empty stack):

```
while expression has tokens {
    if next token is operand /* e.g., number */
          push it on the stack;
    else /* next token should be an operator */
    {     pop two operands from stack;
          perform operation;
          push result of operation on stack; }
}
pop the result; /* should be only thing left on stack */
```

# Postfix evaluation example

- Expression: `5 4 + 8 *`
  - Step 1: `push 5`
  - Step 2: `push 4`
  - Step 3: `pop 4, pop 5, add, push 9`
  - Step 4: `push 8`
  - Step 5: `pop 8, pop 9, multiply, push 72`
  - Step 6: `pop 72` – the result
- A bad postfix expression is indicated by:
  - More than one value on stack at end
  - Less than two operands to pop when operator occurs

# Evaluating infix expressions

- Simplest type: fully parenthesized
  - e.g., ( ( ( 6 + 9 ) / 3 ) * ( 6 - 4 ) )
- Still need 2 stacks: 1 numbers, 1 operators

```
while tokens available {
    if (number) push on number stack;
    if (operator) push on operator stack;
    if ( '(' ) do nothing;
    else { /* must be ')' */
       pop two numbers, and one operator;
       calculate; push result on number stack;
    }
} /* should be one number left on stack at end: the result */
```

# Converting infix to postfix

- Operator precedence matters
  - e.g., `3+(10–2)*5` → `3 10 2 - 5 * +`
- Algorithm uses one stack; prints results
  (alternatively, could append results to a string)
  - For each token in the expression:

```
if ( number ) print it;
if ( '(' ) push on stack;
if ( ')' )
    pop and print all operators until '(';
    discard '(';
if ( operator ) /* more complicated – next slide */
```

# Infix to postfix (cont.)

/* call current token the "new operator" */

```
while (stack is not empty)
   peek at top operator on stack;
   if (top operator precedence >= new operator)
       pop and print top operator;
   else break out of while loop;
push new operator on stack after while;
```

– At end, pop and print all remaining operators. Done.

Notice: We don't know how a stack is implemented yet, but that doesn't seem to matter. Does it?

**Abstraction is good!!!**

# Stack interface

```
interface Stack {
   boolean isFull(); // true iff stack is full
   boolean isEmpty(); // true iff stack is empty
   void clear(); // makes the stack empty
   void push(Object item); // inserts item
            // pre-condition: !isFull()
   Object pop(); // removes/returns last item pushed
            // pre-condition: !isEmpty()
   Object peek(); // just returns last item pushed
            // pre-condition: !isEmpty()
}
```

# Implementing stacks by arrays

- Idea is to keep track of "top" array index
  - `ArrayStack(int capacity)` // constructor –
    `Object array[] = new Object[capacity];`
    `int top = 0;` // some prefer -1 – differences unimportant
  - `isEmpty()` – `return top == 0;`
  - `clear()` – set `top = 0;`
  - `push(Object item)` – `array[top++] = item;`
  - `pop()` – `return array[--top];` // notice *pre*-decrement
  - `peek()` – `return array[top-1];` // no decrement
- Very efficient, but stack is full when array is full
  - `isFull()` – `return top == array.length;`
  - Can use dynamic array, or even better – use ArrayList

# A stack can *adapt* an ArrayList

- No need to keep track of top – let list do that
  - `ArrayListStack()` // no capacity variable either
    `ArrayList list = new ArrayList();`
  - `isEmpty()` – `return list.isEmpty();`
  - `clear()` – `list.clear();`
  - `push(Object item)` – `list.add(item);`
  - `pop()` – `return list.remove(list.size()-1);`
  - `peek()` – `return list.get(list.size()-1);`
- Never full, but slightly less efficient – method overhead
  - `isFull()` – `return false;`
- Note: or with a LinkedList – usually top is *first* element

# Notice what doesn't matter

- `void method(Stack stack) { }`
  - Is it an ArrayStack? ArrayListStack? Other?
  - Use the same way no matter how implemented
- Implementation does affect efficiency – time and space requirements
- Also can affect usefulness (e.g., can get full or not)
- But implementation can be changed
  - Without any changes to client code!
  - Remember to recompile though

# Stack operation complexity

- Implementing a stack with an array
  - peek(), pop() – access last item (remove for pop)
    - Complexity is O(1) – does not depend on n
  - push(object) – add a last item
    - O(1) if array is not full; otherwise O(n) to resize/copy
- Implementing with single-linked list
  - peek(), pop() – access *first* item – Why not last item?
    - O(1) – but would be O(n) if "top" is last item instead
  - push(object) – add a first item
    - Also O(1)
- So same in terms of speed – but different space requirements, and different constants/effort

# What is a recursive method?

- Ans: a method that calls itself (maybe indirectly)
- Standard first example – factorial method:

  $n! = n * (n-1) * (n-2) * \ldots * 1$ (for $n > 0$)

  – Note *recursive* pattern:

  $n! = n * (n-1)!$ (for $n > 1$, and $1! = 1$)

  – Translates immediately to Java:

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

# Recursive solution essentials

- Always need a base case
  - a.k.a. trivial case, or smallest case
  - A way to stop; otherwise infinite recursion
    - e.g., `if (n <= 1)` in factorial method
- Recursive calls converge on base case
  - i.e., problems get smaller with each recursion
    - e.g., `factorial(n-1)`
- Solution must actually solve the problem!
  - This part is most important, and the hardest to insure

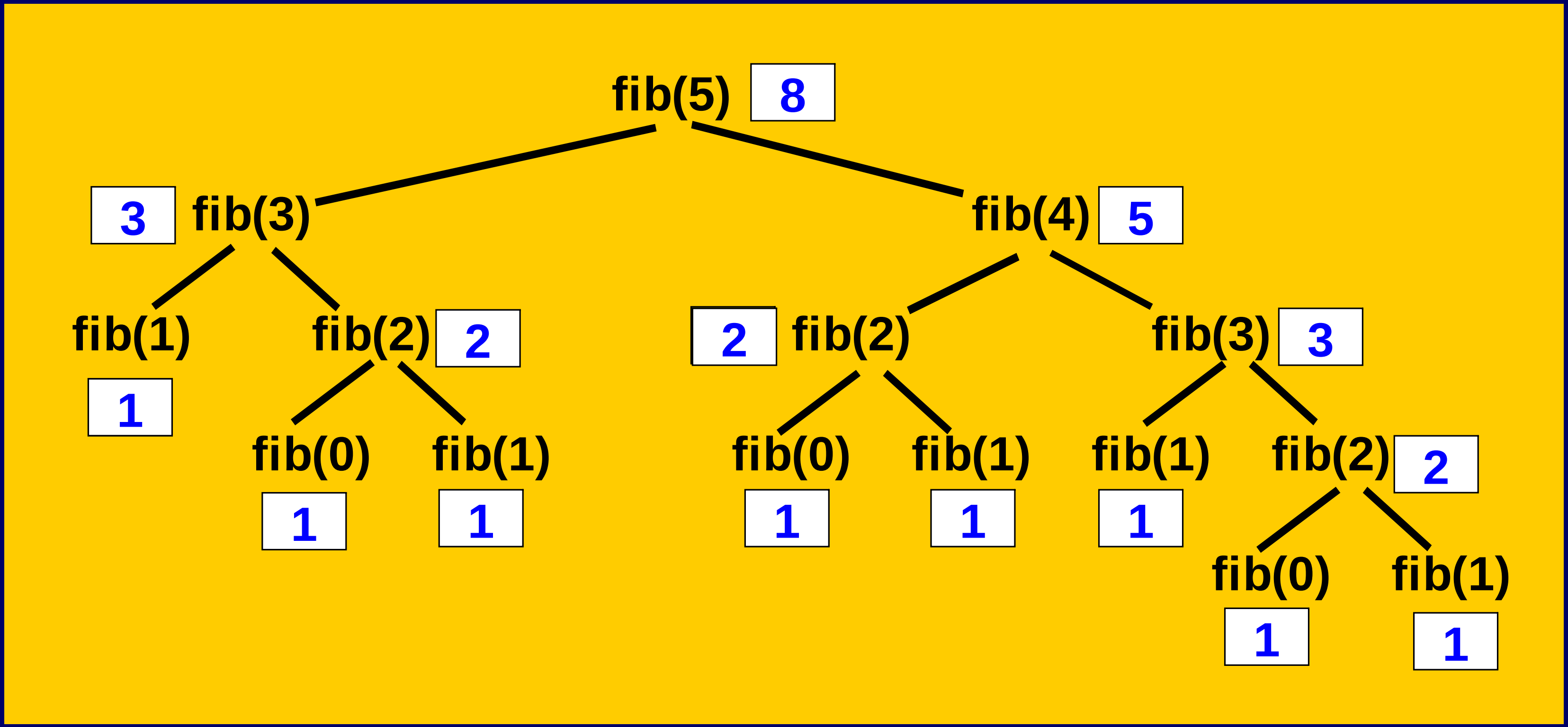
# Fibonacci – a good example, but a poor application

- fib(n) = fib(n-2) + fib(n-1),
  fib(0) = fib(1) = 1
  - Note: general solution has two recursive calls
  - Okay, but in this case, recursion is very inefficient!

    fib(5) calls fib(3), fib(3) calls fib(1),
    fib(3) calls fib(2), fib(2) calls fib(0),
    fib(2) calls fib(1)

    fib(5) calls fib(4), …
  - Count increases exponentially – 15 calls for fib(5), 987 calls for fib(15), 2,692,537 calls for fib(30), …

# fib(5) – call tree




- fib(5) and fib(4) once each, fib(3) twice
- fib(2) 3 times, fib(1) 5 times, fib(0) 3 times

# Recursive Drawing Example

- Handy for some non-numerical problems too
- Drawing tick marks on a ruler:
  - base case: draw nothing (tick too small)
  - general case: draw middle tick, then draw left and right "sub-rulers" (with smaller ticks)

```
void ruler(int left, int right, int tickHeight) {
   if (not done yet) {    /* pseudocode */
      int middle = (left + right) / 2;
      draw_tick(middle, tickHeight);
      ruler(left, middle, tickHeight / 2);
      ruler(middle, right, tickHeight / 2);
   }
}
```

# Maze example

- Suppose we are in a grid-like maze, and need to find an exit
- At each step – can move one square in either of four directions, any of which may be blocked
- Q: how can we use recursion?
  - Key is to find "smaller" problem
- A: *assume we know how* to get to an exit from one of the neighboring squares!

# Recursive maze exit finder

- findExit(x,y) returns true if exit is reachable from maze coordinate (x,y)

```
boolean findExit(int x, int y)  /* first try */
{  if ( x,y is an exit)
       return true;      /* success! */
   if (findExit(x+1, y)  return true;
   else if (findExit(x-1, y) return true;
   else if (findExit(x, y+1) return true;
   else if (findExit(x, y-1) return true;
   else return false;  /* there's no way out of here */ }
```

- Base case? Smaller case? General solution?

**OK** **Not really** **OK**