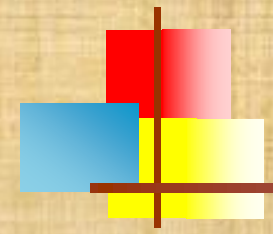
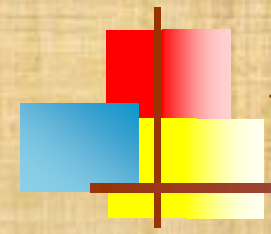
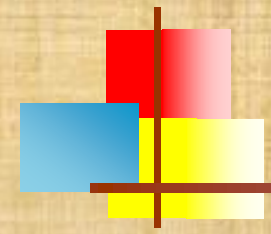


STACK



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



Fundamental stack operations

stack.push(object)

- Adds the object to the top of the stack; the item pushed is also returned as the value of push

```
object = stack.pop(); // object is of type "E"
```

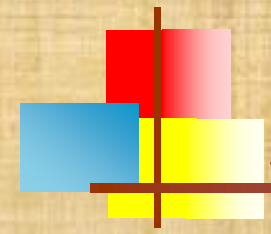
- Removes the object at the top of the stack and returns it

```
object = stack.peek(); // object is of type "E"
```

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

stack.empty()

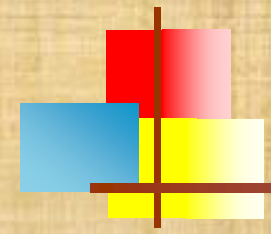
- Returns true if there is nothing in the stack



Additional stack operation

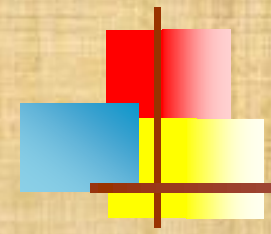
```
int i = stack.search(object);
```

- 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



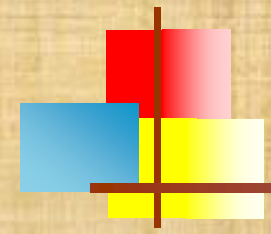
Some uses of stacks

- 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)



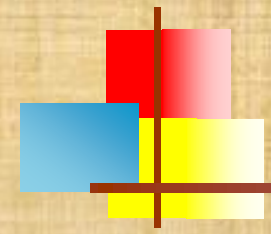
Expression evaluation

- 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



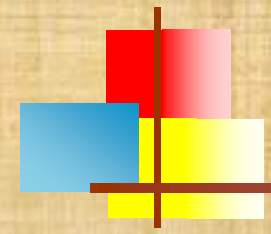
Performing calculations

- 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



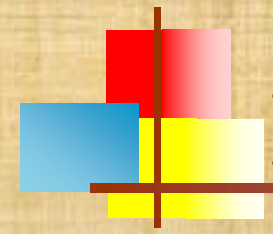
Example: 1+2*3+4

- 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



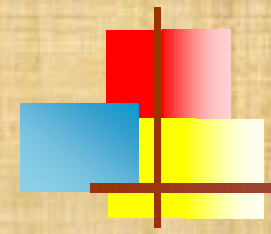
Handling parentheses

- 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



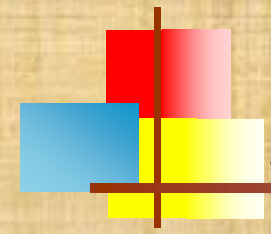
Handling variables

- 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`



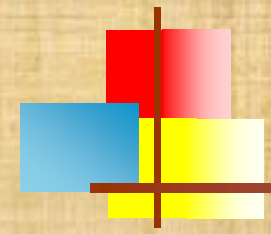
Handling the = operator

- 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