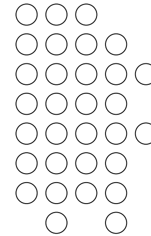
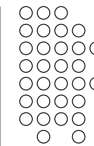
More Linear Data Structures

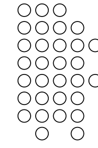
3A

Stacks



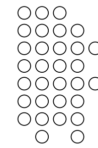
Stacks



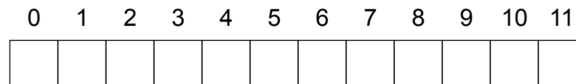


Stack Operations

```
public interface LIFOStack<E> {
  public void push(E element);
      // Insert element at top of stack
  public boolean isEmpty();
      // Is the stack empty?
  public E pop();
      // Remove element from top of stack
  public E peek();
      // Examine element at top of stack
}
```

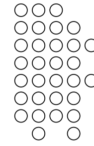
Stacks using an array



Store the elements of the stack from top to bottom in order in the array.

Which end should be the top of the stack? Why?

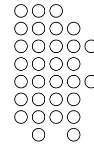
Array Implementation
Fields



```
public class ArrayStack<E> implements
  LIFOStack<E> {

  private E[] dataArray;
  private int top;

  // methods (next slides)

}
```

index of top stack element in array

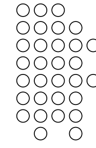
Array Implementation
Constructor & isEmpty



```
  public ArrayStack<E>() {
      dataArray = (E[])new Object[1];
      top = -1;
  }

  public boolean isEmpty() {
      return (top == -1);
  }
```

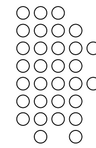
indicates an empty stack

Array Implementation
push



```
public void push(E element) {
    if (____________________________)
         reallocate();      // not shown
    top = top + 1;
    dataArray[top] = element;
}
```

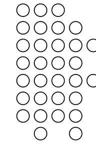

Array Implementation
pop



```
public E pop() {
    if (____________________________)
      throw new NoSuchElementException();
    E element = dataArray[top];
    top = top - 1;
    return element;
}
```

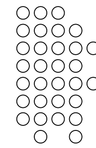
Array Implementation

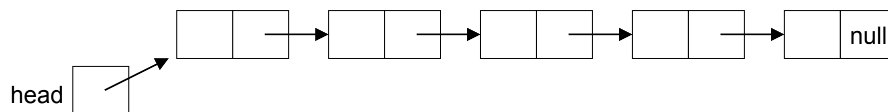
peek



```
public E peek() {
    if (____________________________)
      throw new NoSuchElementException();
    return dataArray[top];
}
```


Stacks using a singly-linked list

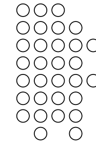




Store the elements of the stack from top to bottom in order in the list.

Which end should be the top of the stack? Why?

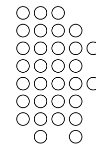
Linked List Implementation
Fields



```
public class ListStack<E> implements
  LIFOStack<E> {

  private Node<E> top;

  // methods (next slides)

}
```

reference to node with top stack element in list

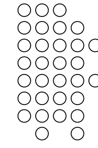
Linked List Implementation
Constructor & isEmpty



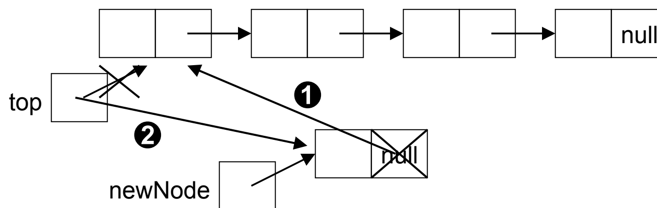
```
  public ListStack<E>() {
      top = null;
  }

  public boolean isEmpty() {
      return (top == null);
  }
```

indicates an empty stack

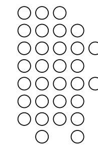
Linked List Implementation
push



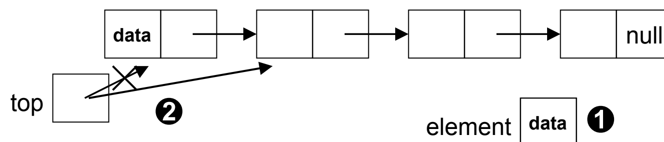
```
public void push(E element) {
    Node<E> newNode = new Node<E>(element);
  ❶ newNode.next = top;
  ❷ top = newNode;
}
```



Linked List Implementation
pop

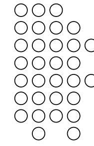


```
public E pop() {
    if (top == null)
      throw new NoSuchElementException();
  ❶ E element = top.data;
  ❷ top = top.next;
    return element;
}
```



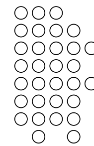
Linked List Implementation

peek



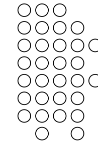
```
public E peek() {
    if (top == null)
      throw new NoSuchElementException();
    return top.data;
}
```


Nested Parentheses

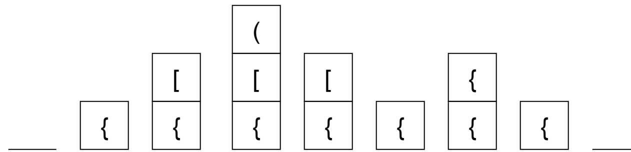


- Determine if a string containing a mathematical expression is nested correctly.
- Parenthetical symbols include { } , [ ], ( ).
- Algorithm:
  - Process the expression from left to right.
    - Push each left parenthetical symbol on a stack.
    - When you encounter a right parenthetical symbol, pop the stack to see if there is a matching left parenthetical symbol. If not, the parenthetical nesting is invalid.
  - After the entire expression is processed, if the stack is empty, the parenthetical nesting is valid.

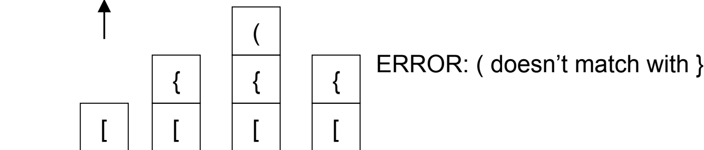
Examples



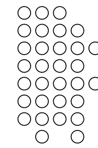
- { [( )] { } }



- [{ (})]



RPN (Postfix Notation)



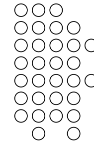
- Some modern calculators use Reverse Polish Notation (RPN)
  - Developed in 1920 by Jan Lukasiewicz
  - Computation of mathematical formulas can be done without using any parentheses
  - Example:
    `5 * ( 6 + 7 )`
    becomes in RPN:
    `5 6 7 + *`

Hewlett-Packard
35s scientific calculator

Hewlett-Packard
12c financial calculator

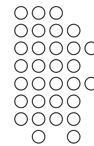
Evaluating postfix with a stack

NOTE: Assume the postfix expression is valid.



1. Let each entry in a postfix expression be a *token*.
2. Let S be an empty stack.
3. For each token in the expression:
   a. If the token is a number, push it on stack S.
   b. Otherwise (the token is an operator):
      i. Pop a token off stack S and store it in y.
      ii. Pop a token off stack S and store it in x.
      iii. Evaluate: x operator y.
      iv. Push result on stack S.
4. Pop the stack for the final answer.

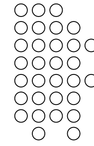
Evaluating postfix: Example



```
Postfix (RPN):   5 7 + 8 6 - / 3 9 * +
```

Converting Infix to Postfix

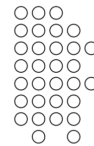
NOTE: Assume the infix expression is valid.



1. Let each entry in an infix expression be a *token*.
2. Let S be an empty stack.
3. Let postfix be an empty string.
4. For each token in the infix expression:
   a. If the token is a number, append it to the postfix string.
   b. Otherwise (the token is an operator), process the operator (**see next 2 slides**)
5. While the stack S is not empty:
   a. Pop an operator off of stack S and append it to the postfix string.

Processing the operators

(Converting infix to postifix, cont'd)

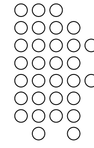


- Each operator has a precedence.
  - *, / multiplicative higher precedence
  - +, - additive lower precedence
- Operators of the same precedence are evaluated left to right.
- Assign precedence values to each operator:

Operator	Precedence
*	2
/	2
+	1
-	1

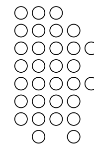
Process the operator

(Converting infix to postfix, cont'd)



1. Let nextOp represent the operator token.
2. If stack S is empty, push nextOp on stack S.
3. Otherwise:
   - a. Let topOp = peek(S).
   - b. If prec(nextOp) > prec(topOp), push nextOp on stack S.
   - c. Otherwise:
     - i. While S is not empty and prec(nextOp) ≤ prec(topOp):
       - a) Let topOp = pop(S).
       - b) Append topOp to postfix string.
       - c) If stack S is not empty, let topOp = peek(S).
     - ii. Push nextOp on stack S.

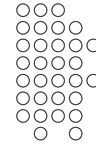
Converting: An example



`Infix:   5 + 4 * 3 / 2 - 1`

Processing the operators

(Converting infix to postfix, cont'd)

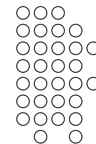


- What if the infix expression contains parentheses?
  - Each pair of parentheses marks a subexpression that must be completely converted before previous operators are processed.
  - Assign a very low precedence to left and right parentheses so only a right parenthesis can pop off a left parenthesis.
  - Do not put parentheses in the postfix string.

Operator	Precedence
*	2
/	2
+	1
-	1
(	-1
)	-1

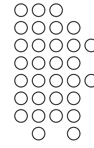
Process the operator - revised

(Converting infix to postfix, cont'd)



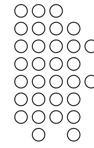
1. Let nextOp represent the current operator token.
2. If stack S is empty **or nextOp is '('**, push nextOp on stack S.
3. Otherwise:
   a. Let topOp = peek(S).
   b. If prec(nextOp) > prec(topOp), push nextOp on stack S.
   c. Otherwise:
      i. While S is not empty and prec(nextOp) ≤ prec(topOp):
         a) Let topOp = pop(S).
         b) **If topOp is '(', exit loop immediately.**
         c) Append topOp to postfix string.
         d) If stack S is not empty, let topOp = peek(S).
      ii. **If nextOp is not ')',** push nextOp on stack S.

Converting: An example

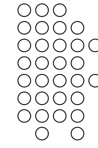


```
Infix:   7 + 5 * ( 4 / 2 + 1 ) - 6
```


Invalid expressions

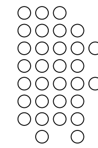


- Assume that tokens are only operators and integers.
- Evaluating a postfix expression:
  - How would you know that the postfix expression was not valid?
- Converting an infix expression to postfix:
  - How would you know that the initial infix expression was not valid?



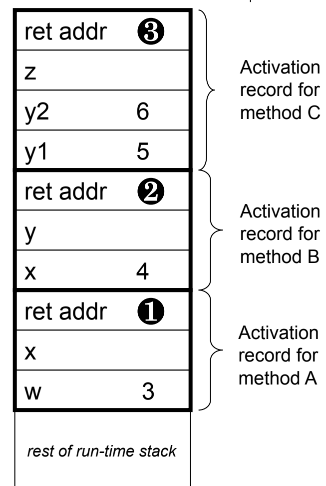
Run-Time Stack & Activation Frames

- Java maintains a *run-time stack* during the execution of your program.
- Each call to a method generates an *activation frame* that includes storage for:
  - arguments to the method
  - local variables for the method
  - return address of the instruction that called the method
- A call to a method pushes an activation record on the run-time stack.
- A return from a method pops an activation record from the run-time stack.

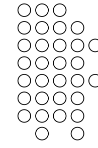


Run-Time Stack & Activation Frames

```
❶ System.out.println(A(3));

public int A(int w) {
❷ int x = B(w+1);
  return x*4;
}
public int B(int x) {
❸ int y = C(x+1, x+2);
  return y*3;
}
public int C(int y1, int y2) {
  int z = y1 + y2;
  return z*2;
}
```



Other uses for stacks



- Computing a convex hull
- Backtracking through a problem space (e.g. maze)
- Parsing algorithms in a compiler
- Machine language architecture
  - Java Virtual Machine