Lecture
Chapter 7
Stacks

Stack

- **LIFO**
- **Top of the Stack**
- **Bottom of the Stack**

Queue

- **FIFO**
- **Front of the Stack**
- **Back of the Stack**

Stack Operations

- **Create an empty stack**
- **Determine whether a stack is empty**
- **Push a new item onto the top of the stack**
- **Retrieve the most recently added item (identify the item, do not remove)**
- **Pop from the stack the item that was most recently added**
    - **Retrieve the most recently added item (identify the item, do not remove)**
    - **Remove the item from the stack**
- **Remove all items from the stack**

UML

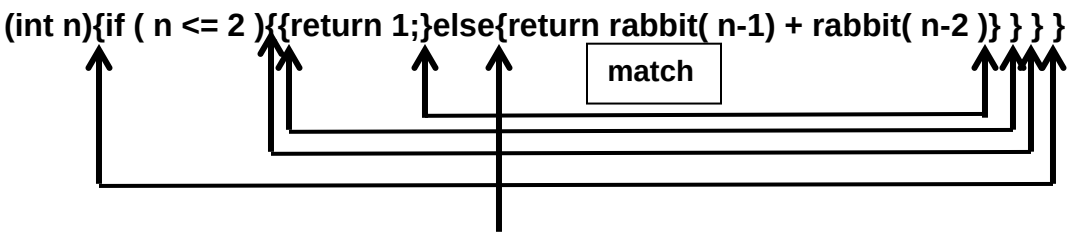
Stack
Top **Items**_____
+createStack() **+isEmpty(): Boolean {query}** **+push(in: newItem:StackItemType) throws StackException** **+pop():StackItemType throws StackException** **//Retrieves & Removes item on top of the stack** **+popAll()** **+peek():StackItemType {query} throws StackException** **//Retrieves, but <u>does not remove</u> item on top of the stack**

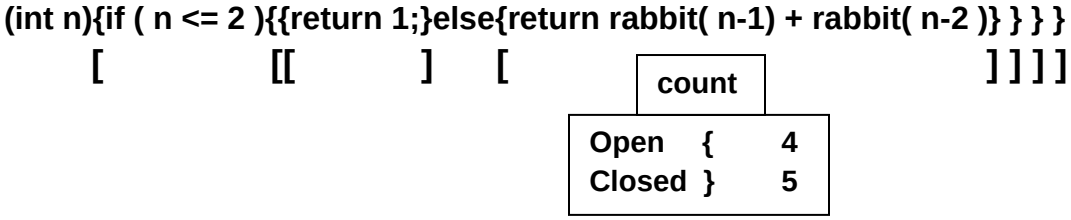
Preconditions
Postconditions
Axioms

1. **(aStack.push(newItem)).pop() == aStack**

Balanced Braces

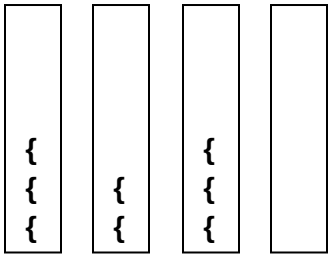
```
public static int rabbit(int n){if ( n <= 2 ){{return 1;}else{
return rabbit( n-1) + rabbit( n-2 )} } } }
```





For each "{" encountered
Push "{" on the stack

For each "}" encountered
Pop a "{" off the stack



Unbalanced since have encountered another "}" with no "{" to pop from the stack

```
aStack.createStack()
balancedSoFar = TRUE
i = 0

while( balancedSoFar & i < aString.length() )
{
  ch = char in ith position in aString
  ++i
  if (ch == "{" )
  {
            aStack.push( "{" );
  }
  else if (ch == "}" )
  {
    if ( ! aStack.isEmpty() )
    {
      openBrace = aStack.pop()
    }
    else
    {
      balancedSoFar = FALSE
    }
  }
}

if ( balancedSoFar & aStack.isEmpty() )
{
  aString balanced
}
else
{
  aString unbalanced
}
```

```
else if (ch == "}" )
{
  try
  {
    openBrace = aStack.pop()
  }
  catch (StackException e )
  {
    balancedSoFar = FALSE
  }
}
```

String Recognition in Languages

L = { w$w' : w != $, w may be empty }

ABC$CBA ∈ L, AB$AB & ABC$CB ARE not in L

```
aStack.createStack()

i = 0
ch = char at position i in aString
while (ch != "$")
{
  aStack.push(ch)
  ++ i
   ch = char at position i in aString
}

++i    // skip "$" character
inLanguage = TRUE

while(inLanguage & i < aString.length)
{
   ch = char at position i in aString
   try
   {
     stackTop = aStack.pop()
     if (stackTop == ch)
     {
        ++i
     }
     else
     {
       inLanguage is FALSE       // characters do NOT match
     }
   }
   catch (StackException e)
   {
      inLanguage is FALSE
   }
}
if (  inLanguage & aStack.isEmpty( ))
            aString is in the Language
else
            aString is NOT in the Language

}
```

Implementations -- Abstract Data Type -- Stack

```
public interface StackInterface
{
   public boolean isEmpty( );
   public void popAll( );
   public void push ( Object newItem ) throws StackException;
   public Object pop( ) throws StackException;
   public Object peek( ) throws StackException;
}
```

For Preconditions & Postconditions see the textbook

```
public class  StackException extends java.lang.RuntimeException
{
  public StackException ( String s )
  {
     super (s);
  }
}
```

Methods that throw StackException do not have to be enclosed in try-catch blocks

Array-Based Implementation

```
public class StackArrayBased implements StackInterface
{
   final int MAX_STACK = 50;
   private Object items[ ];
   private int top;

   public StackArrayBased( )
   {
       Items = new Object[ MAX_STACK ];
       top = -1;
   }

   public boolean isEmpty( )
   {
      return top < 0;
   }

   public boolean isFull( )
   {
      return top == MAX_STACK – 1;
   }
```

Defining the array "items" and the "top" variable as private secures the Array Based Stack's abstraction, i.e., the "wall" are secure. The use of the StackException provides an easy way to control attempts to violate the Stack ADT protocols

The items array hold only Objects so if you want to store integers in the stack they must be placed in their wrapper classes, e.g., Integer for int, etc.

See textbook for the remaining methods required to implement a stack using an array

```
}
```

Reference-Based Implementation

```
public class StackReferenceBased implements StackInterface
{
    private Node top;

    StackReferenceBased ( )
    {
       top = null;
    }


    public Boolean isEmpty ( )
    {
       return top == null;
    }

    public void push ( Object newItem )
    {
       top = new Node( newItem,  top );
    }
```

See textbook for the remaining methods required to implement a stack using linked lists

```
}
```

List-Based Implementation

```
public interface ListInterface
{
   public boolean isEmpty( );
   public int size( );
```

See textbook for the remaining methods required to define the interface (page 265, 3rd ed)

```
}

public class ListReferenceBased implements ListInterface
{
    private Node head;
    private int numItems;
```

See textbook for the remaining methods required to define the ListReferenceBased (page 265, 3rd ed)

```
}

public class StackListBased implements StackInterface
{
  private ListInterface list;

  public StackListBased ( )
  {
     list = new ListReferenceBased ( );
  }

  public boolean isEmpty( )
  {
     return list.isEmpty( );
  }

   public void push(Object newItem)
   {
      List.add(0, newItem);
   }
```

See textbook for the remaining methods required to define the StackListBased (page 370, 3rd ed)

```
}
```

See textbook for Comparing Implementations (page 371, 3rd ed)

Java Collections Framework

- **Interface List**
- **Class Stack**

```
public class Stack<E> extends Vector<E>
{
    public Stack( );
    public boolean empty( );
    public E peek( ) throws EmptyStackException;
    public E pop( ) throws EmptyStackException;
    public E push (E item);
    public int search(Object o);
}
```

```
import java.util.Stack;

public class TestStack
{
    static public void main( String [ ] args )
    {
       Stack<Integer> attack = new Stack<Integer>( );

       See textbook for the remaining methods required to
       complete the program   (page 372-3, 3rd ed)

    }
}
```

Evaluating Postfix Expressions

Operands ➔ push
Operators ➔ pop last two operands, apply operator to the operands, push result

234+* ➔ push 2, push 3, push 4, pop 4, pop 3, apply + to 3 & 4, push 7
27* ➔ pop 7, pop 2, apply * to 2 & 7 ➔ push 14

Converting Infix Expressions to Postfix Expressions

- **Operands always stay in the same order with respect to one another**
- **Operators move only to the right of the operands**
- **Parentheses are removed only when no longer needed**

String postfixExp = null;

If you encounter

- **an operand on the infixExp string, append it to the postfixExp**
- **"(" on the infixExp string, push it on the stack**
- **an operator on the infixExp string,**
  - **if stack is empty, push the operator on to the stack**
  - **if stack is NOT empty, pop operators of <u>greater or equal precedence</u> from the stack and append them to the postfixExp;**
    STOP when you encounter either
    - **a "(" or**
    - **an operator of lower precedence, or**
    - **when the stack is empty**

    then <u>push the operator on to the stack</u>
- **")"on the infixExp string,**
  - **pop operators off the stack &**
  - **append them to the end of the postfixExp until you encounter a matching "("**
- **the end of the infixExp, append the remaining contents of the stack to postfixExp**

See textbook for the pseudocode algorithm which converts infix expressions to postfix expressions (page 378, 3rd ed)

See textbook for two solutions to a graph based search problem (pages 378-379, 3rd ed),

- **one using stacks (pages 380-388, 3rd ed) and**
- **the other using recursion (pages 388-391, 3rd ed)**