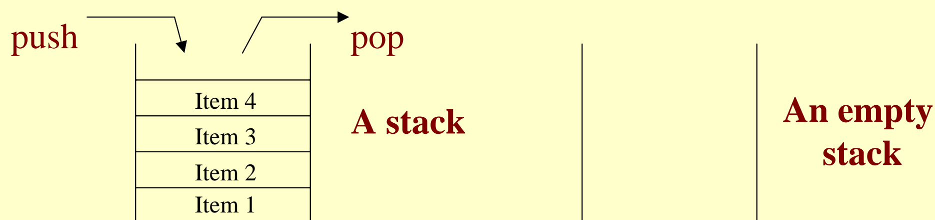
Course: Programming II - Abstract Data Types

The ADT Stack

Definition

- The **ADT Stack** is a linear sequence of an arbitrary number of items, together with access procedures.
- The access procedures permit **insertions** and **deletion** of items only at one end of the sequence (the "**top**").
- The stack is a list structure, sometimes called a **last-in-first-out** (or LIFO) list.
- A stack is either empty, or it consists of a sequence of items. Access is limited to the "top" item on the stack at all times.



The ADT Stack and Recursion

Slide Number 1

The term "stack" is intended to conjure up visions of things encountered in daily life, such as a stack of dishes in a college cafeteria or a stack of books on your desk. In common English usage, "stack of" and "pile of" are synonymous. To computer scientists, however, a stack is not just any old pile. A stack has the property that the last item placed on the stack will be the first item to be removed. This property is commonly referred to as "last-in-first-out" or simply **LIFO**. The last item placed on the stack is called the "**top**" item in the stack. Access procedures for this type of Abstract Data type can therefore only examine the top item.

The LIFO property of stack seems inherently unfair. How would you like to be the first person to arrive on the "stack" for a movie (as opposed to a line for a movie). You would be the last person to be allowed in! Stack are not especially prevalent in everyday life. The property that we usually desire in our daily lives is "first-in-first-out" (or FIFO). A **queue**, which we will study in the next lecture, is the Abstract Data Type with the FIFO property. Most people would much prefer to wait in a movie queue (or in a line) than in a movie stack! However, while the LIFO property is not very appropriate for many day-to-day situations, it is very much needed for a large number of problems that arise in computer science.

The access procedures for a stack include therefore operations such as examining whether the stack is empty (but not how many items are in the stack), inspecting the top item but not others, placing an item on top of the stack, but at no other position, and removing an item from the top of the stack, but from no other position. Stacks can therefore be seen as special lists with access procedures restricted to just the top element.

Access Procedures

Constructor operations to construct a stack:

i. **createStack( )**
// post: creates an empty Stack

ii. **push(newItem)**
// post: adds newItem at the top of the stack.

Predicate operations to test Stacks

i. **isEmpty( )**
// post: determines whether a stack is empty

Selector operations to select items of a Stack

i. **top( )**
// post: returns the top item of the stack. It does not change the stack.

ii. **pop( )**
// post: changes the stack by removing the top item.

This slide provides the headers of the access procedures for the ADT stack, and their respective post-conditions, without the exception cases. The names "push" and "pop" given here for the operations that add and remove items to the stack are the conventional names for stack operations.

So far we have only given the headers of these operations and their (partial) post-conditions. Later we will give examples of implementations. In the next slide we'll see a brief example of a program application that uses this ADT.

The full post-conditions for these access procedures that include also the exception cases are given here:

top( )
//post: if the stack is not empty, the item on the top is returned
//and the stack is left unchanged. Throws exception if the stack is empty.

pop( )
//post: if the stack is not empty, the item on the top is removed from the
//stack. Throws exception if the stack is empty

push(item)
//Pre:item is the new item to be added
//Post: If insertion is successful, item is on the top of the stack.
//Throws StackException if the insertion is not successful

Using a Stack: an example

A method displayStack that displays the items in a stack:

The pseudocode:

```
displayStack(aStack)
// post: displays the items in the stack aStack;
  while (!aStack.isEmpty( ) ) {
          newChar = aStack.top( ));
          aStack.pop( );
          Write newChar
  } //end while
```


As it is the case for an ADT list, also for stacks once the operations of the ADT stack have been satisfactorily specified, applications can be designed that access and manipulate the ADT's data, by using only the access procedures without regard for the implementation. So far, in fact, we haven't seen or used any particular implementation of stacks.

A little example is considered here. Suppose that we want to display the items in a stack. The "wall" between the implementation of the ADT and the rest of the program prevents the program from knowing how the stack is being stored, i.e. which data structure has been used to implement the stack.

The program can, however, still access the items of a stack by means of the access procedures. In this case the method **displayStack**, can use the operation **aStack.top( )** to access the top item of a stack, and the operation aStack.pop() to change the stack by removing the item on the top.

Course: Programming II - Abstract Data Types

Axioms for ADT Stacks

The access procedures must satisfy the following axioms, where Item is an item and aStack is a given stack:

1. (aStack.createStack( )).isEmpty = true
2. (aStack.push(Item)).isEmpty() = false
3. (aStack.createStack( )).top( ) = error
4. (aStack.push(Item)).top( ) = Item
5. (aStack.createStack( )).pop( ) = error
6. (aStack.push(Item)).pop( ) = aStack

A stack is like a list with top( ) , pop( ) and push(item) equivalent to get 1st item, remove 1st item and add item at the beginning of a list.

The ADT Stack and Recursion Slide Number 4

In this slide I have listed the main axioms that the access procedures for an ADT stack have to satisfy. In this case, we have two axioms for each operation.

Note that the access procedures for a stack are somewhat equivalent to the access procedures for a list described in the previous lecture. In particular, the top( ) procedure for a stack can be seen as equivalent to the get(1) procedure for a list, which takes the first element of a given list. In the same way, the procedure pop( ) for a stack is equivalent to the procedure remove(1), which removes the first element in a list, whereas the procedure push(item) is equivalent to the procedure add(1,item), which add the given new item to the first position in a list.

In the next slides we'll look at the two different types of implementations of a stack, one based on array and the other based on linked lists. The definition of the data structure for a stack in the case of a dynamic implementation, will further illustrate the fact that stacks are essentially lists with specific use of their access procedures.

Interface *StackInterfac*e for the ADT Stack

```
public interface StackInterface{

    public boolean isEmpty();
    //Pre: none
    //Post: Returns true if the stack is empty, otherwise returns false.

    public void push(Object item) throws StackException;
    //Pre:item is the new item to be added
    //Post: If insertion is successful, item is on the top of the stack.
    //Post:Throw StackException if the insertion is not successful

    public Object top( ) throws StackException;
    //Pre: none
    //Post: If stack is not empty, the item on the top is returned. The stack is left unchanged
    //Post:  Throws StackException if the stack is empty.

    public void pop( ) throws StackException;
    //Pre: none
    //Post: If stack is not empty, the item on the top is removed from the stack.
    //Post:  Throws StackException if the stack is empty.
```


The interface StackInterface given in this slide provides the main definition of each access procedure for the ADT stack. As in the case of the ADT list, the constructor createStack( ) is not included here as it is normally given as constructor of the particular Java class that implements the Stack.

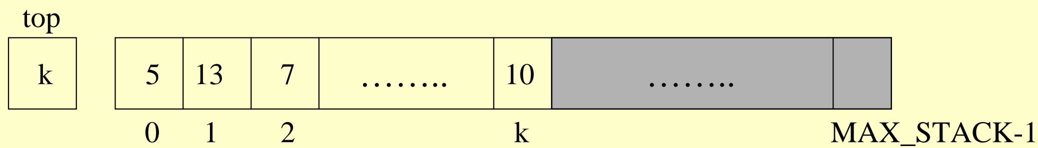
The two main types of implementation (array-based, reference-based) for a stack are classes that implement this interface. This interface provides, therefore, a common specification for the two implementations that we are going to illustrate in the next few slides.

An example of the StackException for a stack can be the following:

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

Note that this exception can be avoided by including in the implementation of the access procedure a check to see whether the stack is empty before calling the procedures. In the case of push(item), if the implementation is a static implementation it is also important to consider exceptions caused by the underlying array data structure being full. A private method, called *isFull*, can be defined for the static implementation of a stack, to check whether the underlying array is full before calling the procedure push(item). In a reference-based implementation, such exception is not necessary since the memory is dynamically allocated and not fixed.

Array-based Implementation of a Stack



Data structure

```
class StackArrayBased{
     final int MAX_STACK = 50;
     private Object items[ ];
     private int top;
…};
```

```
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;
     }// end default constructor;

     public boolean isEmpty( ){
            return top < 0; }// end isEmpty
……}
```


The data structure for a static implementation of a stack uses an array of Objects called **items**, to represent the items in a stack, and an index value **top** such that items[top] is the stack's top.

When a stack is created it does not include any item at all. In this case the value of the index top is set to –1. This allows us to test in general when a stack has become empty by just checking whether the value of top is a negative integer.

The partial implementation of StackArrayBased given in this slide gives the definition of the default constructor and the implementation of the method isEmpty.

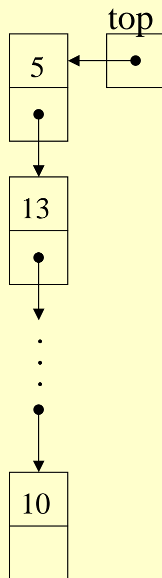
In the same way it would be possible to check whether a stack is full by testing whether the value of top has become equal to MAX_STACK-1.

The addition of items in the stack, starts from position 0 in the array. Each time a push procedure is called, the value of top is incremented by 1 to point to the next top free cell in the array, and the new item is assigned to this position.

Similarl;y, each time a pop( ) procedure is called, the item at the top of the stack should be removed. In this static implementation, this is simply given by decrementing the value of top by 1. The array will still include the value but no longer point to it. An example implementation for the procedure pop( ) is given here:

```
Public void pop( ) throws StackException{
        if ( !isEmpty( ) ) { top = top –1;}
        else {
              throw new StackException("Exception on pop: stack empty");
        }
}
```

Dynamic Implementation of a Stack



Data structure

```
class StackReferenceBased{
    private Node top;
…}
```

```
class Node{
    Object item;
    Node next;
…}
```

Implementation

```
public class StackReferenceBased
                    implements StackInterface{

    private Node top;

    public StackReferenceBased( ){
        top = null;
    }// end default constructor;

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


Many applications require a reference-based (or dynamic) implementation of a stack so that the stack can grow and shrink dynamically. A diagrammatical representation of a dynamic implementation of a stack is given in the left-hand side of this slide. In this picture, top is a reference to the head of a linked list of items. **The implementation uses the same node class used for the linked list in the previous lecture.**

We give here the implementation of the other access procedures:

```
public void push(Object newitem){
        top = new Node(newitem, top);
}// end push


public Object top( ) throws Stack Exception {
        if (! isEmpty)  {
          return top.getItem( ); }
        else throw new StackException("Stack is empty");
}

public void pop( ) throws StackException {

        if (!isEmpty) {
            top = top.getNext;
}
```

Recursion

Java uses stacks to implement method activations and recursion.
General criteria:

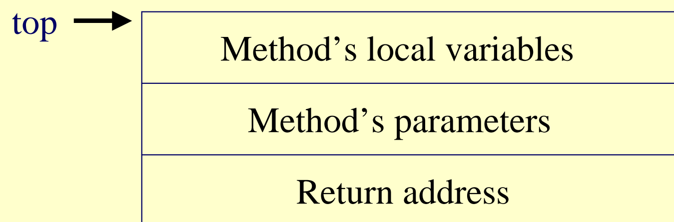
- A representation of each variable declared within a method is created on entry to (or activation of) that method, and destroyed on exit from it.
- A distinct representation of each variable is created on each re-entry to (or re-activation of) a method.

In particular, these criteria enable Java to implement recursive methods, by having as many instances of the declared variables in existence as activations have been made of the method.
But only the most recent are accessible to the program.

In the remainder of this lecture we will have a closer look at how methods are executed in Java and in particular how recursion is handled in Java. The reason why this is given together with the introduction to the ADT Stack, is because the underlying data structure used by Java for handling both method activations and recursive calls of a method is in fact a stack.

Activation records and execution stack

An **activation record** is a memory block created at each method activation, which includes:



The return address is information about the correct location to return to after the method has been executed.

An **execution stack** stores the collection of activation records: each activation of a method pushes the next activation record on top of the execution stack.

Java keeps track of the activation of a method (say) A, in the following way. When a method A is activated (whether recursive or not) the computer temporarily stops the execution of the current method (this could well be a main program). Before actually executing the method A, some information is saved that will allow the computer to return to the correct location after the method A is completed. The computer also provides memory for the parameters and local variables that the method A uses. This memory block is called "activation record" and essentially provides all the important information that the method A needs to work. It also provides information as to where the method should return when it is done with its computation. Once this block is created, the method A is then executed. If the execution of method A should encounter an activation of another method, recursive or otherwise, then the first method's computation is temporarily stopped. This is because the second method must be executed first before the first method can continue. Information is saved that indicates precisely where the first method should resume when the second method is completed. An activation record for the second method is then created and placed on top pf the other existing activation records. The execution then proceeds to the second method.. When this second method is completed, its activation record provides the information as to where the computation should continue. The execution record of this second method is then removed from the top of the collection of existing activation records.

All the activation records so created are stored by the program in a stack data structure called "execution stack". Each newly created activation record is pushed on top of the execution stack and at the end of execution of a method, its activation record is popped off the execution stack.

This mechanism is used for both recursive and non-recursive methods. We will now see, in particular, examples of recursive methods.

Iteration and Recursion

Many algorithms have alternative iterative and recursive forms:

E.g.: The Factorial function:

```
int Factorial(int n){
//pre: n≥ 0. It evaluates n! iteratively
int temp=1;
if (n = = 0) then return 1
else {
        for (int i=1; i≤n; i++)
            temp = temp*i;
      }
return temp;
}
```

Iterative algorithm

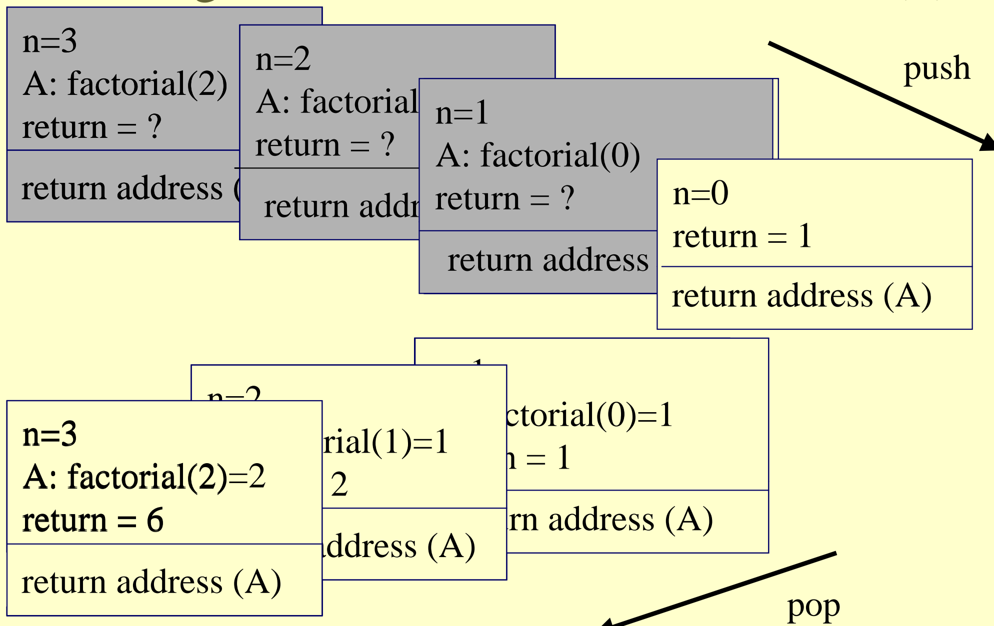
```
int Factorial(int n){
//pre: n≥1. It evaluates n! recursively
if (n = = 0) then return 1
else return n*Factorial(n-1);
}
```

Recursive algorithm

Many algorithms can have either an iterative or recursive form. In this slide I have given the example of the algorithm that calculates the factorial of a natural number in both its two forms.

Let’s see what happens when we run for instance the recursive algorithm for the case of Factorial (3).

Executing the recursive method Factorial(3)



This is an example run of recursive call. Note this slide is animated. At each creation of a new activation record that is pushed in the execution stack, the current activation record becomes grey, as it is temporarily stopped . At the end, on the last recursive call, the operation is a simple return instruction. The last activation record is then destroyed or popped off the stuck. And the previous activation record becomes activated (in the activation it becomes white again)... until we reach the activation record at the bottom of the stuck and the program is then completed.

Example2: The Fibonacci function

```
int Fib(int n){
//pre: n>0. It evaluates Fib(n) recursively
//post: returns the nth Fibonacci number
//      of order 2
if (n ≤ 2) then return 1;
else return (Fib(n-1)+Fib(n-2));
}// end Fib
```

```
int Fib(int n){
//pre: n>0. It evaluates Fib(n) iteratively
//post: returns the nth Fibonacci number
//      of order 2
int i, this, previous, temp;
if (n ≤ 2) then return 1
else {  this = 1;
        previous = 1;
        i  = n;
        do {    temp = previous+this;
                previous = this;
                this=temp;
                i--;
           }
        while (i > 2);
        return this;}
}// end Fib
```


This is another example of an algorithm that can be executed either recursively or iteratively. The algorithm is the function that calculates the nth Fibonacci number of order 2.

You should already know what the Fibonacci numbers are. These are number that form a special sequence defined as follow:

Fib(1) = 1;

Fib(2) = 1;

Fib(n) = Fib(n-2) + Fib(n-1), for any n>2

The recursive algorithm used here is called **binary recursive**, because it makes two recursive calls on smaller problems (i.e. smaller numbers). There are other types of recursive algorithms, but we are not going to go into more detail in this part of the course.. If you are interested you can look at the text book on data structures for more information about recursive algorithms.

Recursion vs Iteration

Recursion provides us with the simplest and most natural solution to a problem. But it can be costly.

Tail recursion is a particular pattern of recursion that can be easily transformed into iteration:

Definition:
A method is tail recursive if there is no code to be executed after the recursive call.

```
public static int listLength(Node head)
{
  if (head = = null) return 0;
  else return 1+ listLength(head.next)
}//end recursive form
```

```
public static int listLength(Node head)
{ Node cursor;
  int answer = 0;
  for (cursor = head; cursor !=null;
              cursor= cursor.next)
        { answer++;}
  return answer;
}//end iterative form
```


Recursion can sometime be costly, even though it might seem to be the most natural solution to a problem. Since each recursive call creates an activation record in the execution stack, if the sequence of recursive calls continues for a long time before the stopping case occurs, then we'll end up with a huge execution stack of activation records. In this case we say that the recursive algorithm is costly in space. It is possible that the size of the execution stack becomes too large for the system resources. In this case Java would issue a "StackOverflowError".

Because of this possibility, it is always important to have some idea of the maximum number of recursive calls needed before reaching a stopping case. This number is called "**depth of recursion**". When the depth is likely to be too big and cause an overflow, the iterative form of the same algorithm should be used instead.

One type of recursion that is easy to eliminate and transform into an iterative form, is "**tail recursion**". This is when there is no code to be executed after the recursive call. An example of tail recursion is given in this slide together with its corresponding iterative algorithm.

In general, a tail recursion has an if statement which checks for the final directly solvable case, followed by a recursive call. To transform this algorithm into its iterative form, we need just to replace the recursive calls with a loop statement and evaluation and assignment of the parameters used in the recursive call.

Another typical example application of recursion is the **Tower of Hanoi**. It's fun. So, if you have some spare time I invite you to look at it in your text book.

Summary

An ADT Stack is:

- A linear sequence of arbitrary number of items, whose access procedures have a last-in, first-out (LIFO) behaviour.
- A strong relationship between recursion and stacks exists. Most implementations of recursion maintain an execution stack of activation records.

Recursion:

- Is a technique that solves a problem by solving a smaller problem of the same type.
- Some recursive solutions are much less efficient than a corresponding iterative solution, due to the overhead of method calls.
- Iterative solutions can be derived from recursive solutions.