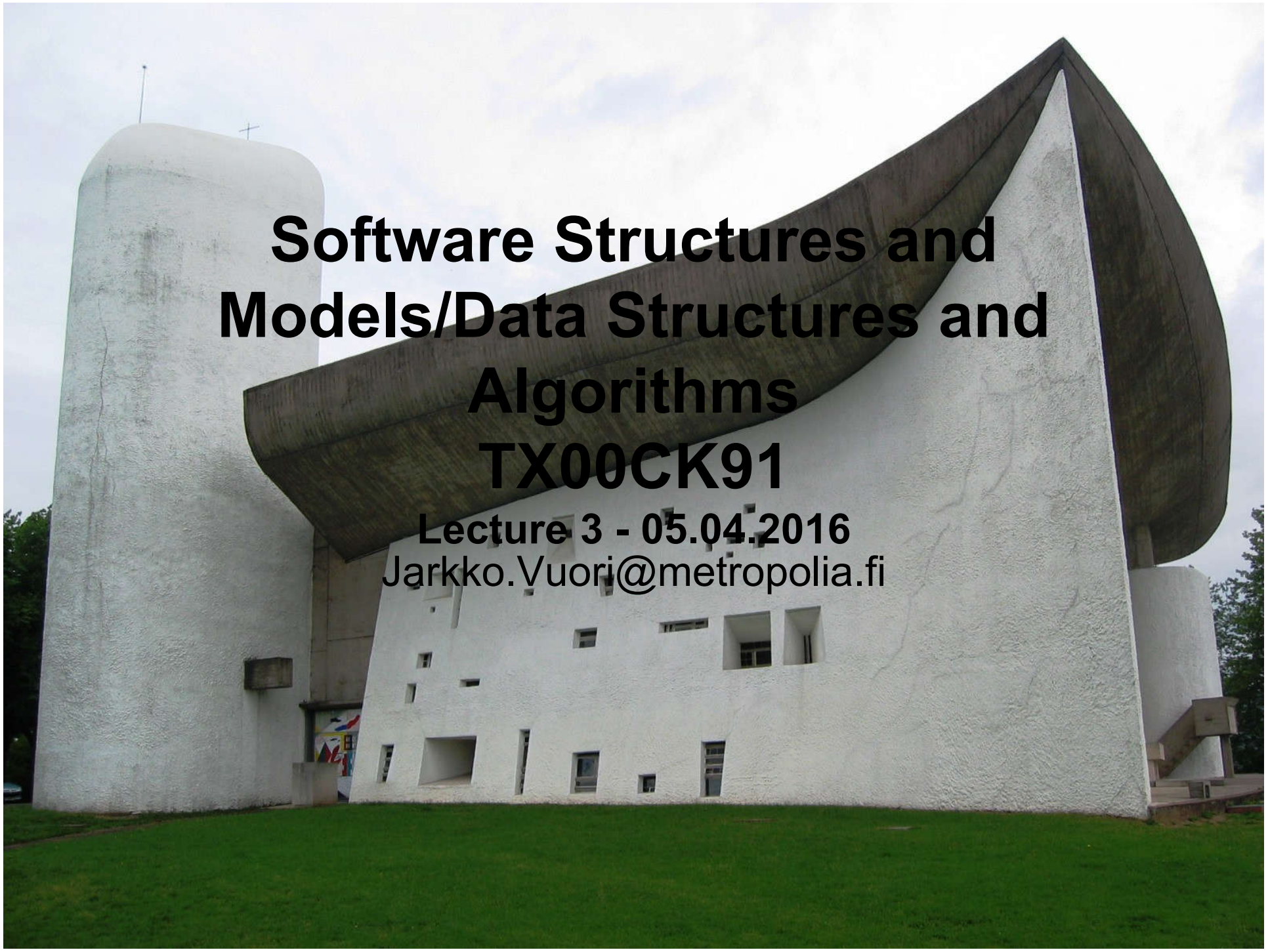


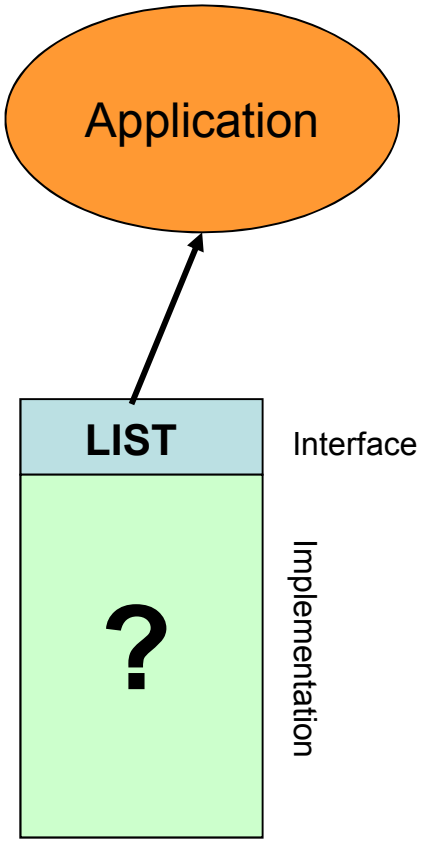
Software Structures and Models/Data Structures and Algorithms TX00CK91

Lecture 3 - 05.04.2016

Jarkko.Vuori@metropolia.fi

Different ways to implement a list

1. Using an array
   - problems
     - We need to move data when new elements are inserted or deleted to/from the list
     - When we ran out of the space (memory), we need to reserve new (larger) space, and copy all the elements from the old space to the new space, and finally release the old space
       - This may happen when inserting an element
2. Linked array
   - Space reserving problem of point 2 still remains
3. Dynamically reserved, linked structure
   - Both problems of point 1 have been removed
   - New problem: element searching takes more time, because direct searching (by index) is not possible
     - inserting an element



Because it is a class, we can freely change the implementation

Different ways to implement a list

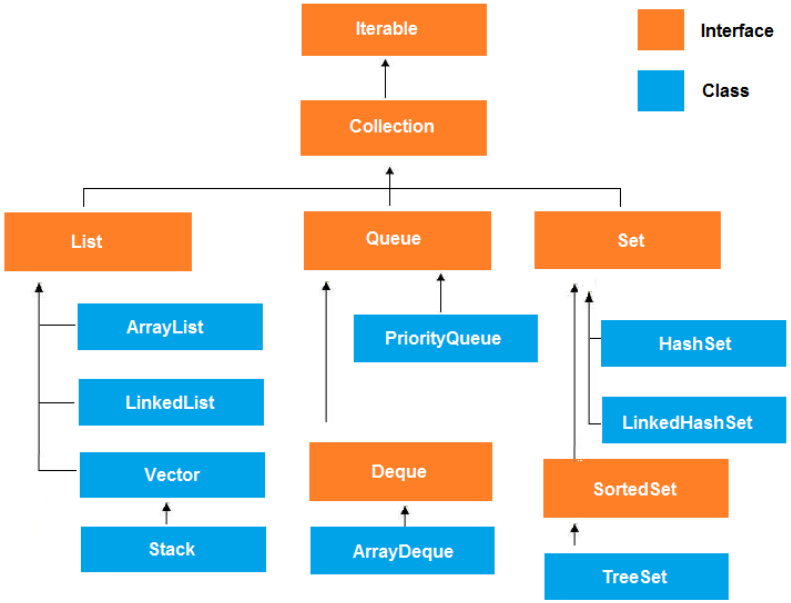
- Each of those different implementations are good in a specific situation
  - Although our goal is to have the maximum generality, totally universal solution is not possible
    - We have similar situation with transportation technology
      - With a sport car (e.g., Lamborghini Diabolo) it is possible to move from one place to other very rapidly and with style - but it is impractical to transfer large amounts of good with a sport car
      - Truck (e.g. Volvo FH16) can be used to transfer large amounts of goods - but it is not very rapid (and there is no style with it)
  - We must ask ourselves considering every application, where it (this particular implementation) is suited and where it is not

Java Collections Framework

- A coupled set of classes and interfaces that implement commonly reusable collection data structures
- Designed and developed primarily by Joshua Bloch (currently professor at Carnegie Mellon University, formerly Chief Java Architect at Google)
  - Also author of extremely recommended book, Joshua Bloch: “Effective Java”, 2nd edition, Addison-Wesley, 2008
- Apart from the Java Collections Framework, the best known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy
- Collection is an object that groups multiple elements into a single unit
  - Sometimes called a container
- Collection Framework is a unified architecture for representing and manipulating
- It Includes:
  - Interfaces: A hierarchy of ADTs
  - Implementations
  - Algorithms: The methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces
    - These algorithms are polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface

Java Collections Framework

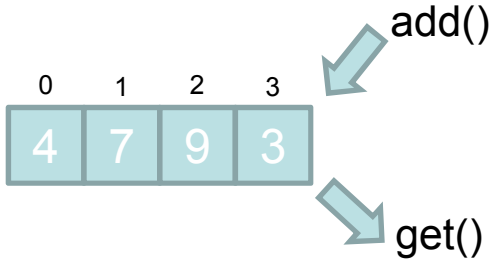
- List Interface
  - A List is a ordered collection that can contain duplicate values. It provides three general purpose implementations
    - ArrayList
      - ArrayList is said to be the best performing list implementation in normal conditions. In simple words we can say that ArrayList is a expendable array of values or objects
    - LinkedList
      - Linked list is a bit slower than ArrayList but it performs better in certain conditions.
    - Vector
      - Vector is also a growable array of objects, but unlike ArrayList Vector is thread safe in nature.
- Set Interface
  - A set is a Interface that does not contain duplicate values. In provides three general purpose implementations in Java
    - HashSet
      - HashSet is the best performing implementation of Set interface. It stores its elements in a HashTable an does not guarantee of any type of ordering in iteration
    - TreeSet
      - TreeSet is a little slow than HashSet and it stores its elements in a red-black tree structure. TreeSet orders its elements on the basis of their values.
    - LinkedHashSet
      - LinkedHashSet is implemented as a HashTable with a LinkedList running through it. It orders its elements on the basis of order in which they were inserted in the set



Container basics

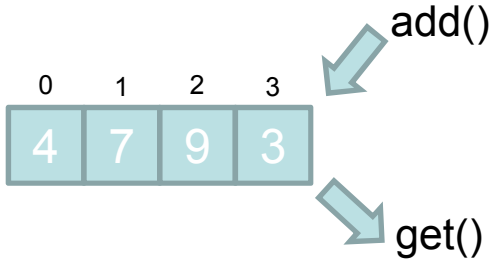
- Containers allow certain types of data to be grouped into logical structures
  - The STL provides generic containers so that the same data structure can handle multiple different types without any additional required coding
- One design principle of Java Collections containers is the effectiveness
- A suitable container should be selected according the effectiveness requirements of different operations applied to the container
- In Java Collection each container has a totally different implementation
  - They are not actually abstract data types where only interface (possible operations) matter
  - For example it is possible to access what ever element in the dequeue
- So-called container adaptors are actually abstract data types
  - Examples of these are stack and queue

ArrayList



- The ArrayList class extends AbstractList and implements the List interface
  - ArrayList supports dynamic arrays that can grow as needed
- Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold
- Array lists are created with an initial size
  - When this size is exceeded, the collection is automatically enlarged
  - When objects are removed, the array may be shrunk
- Constructors
  - ArrayList( )
    - This constructor builds an empty array list.
  - ArrayList(Collection c)
    - This constructor builds an array list that is initialized with the elements of the collection c.
  - ArrayList(int capacity)
    - This constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

ArrayList



- void add(int index, Object element)
  - Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index > size())
- boolean add(Object o)
  - Appends the specified element to the end of this list
- void clear()
  - Removes all of the elements from this list
- boolean contains(Object o)
  - Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e))
- void ensureCapacity(int minCapacity)
  - Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument
- Object get(int index)
  - Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index ≥ size())
- int indexOf(Object o)
  - Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element
- Object remove(int index)
  - Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if index out of range (index < 0 || index ≥ size())
- Object set(int index, Object element)
  - Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index ≥ size())
- int size()
  - Returns the number of elements in this list

ArrayList example

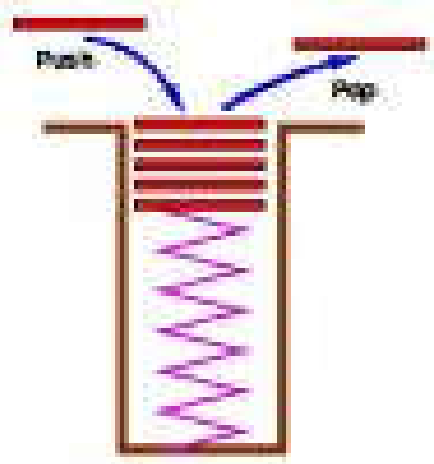
- The example on the right would produce the following result:

Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]

```
import java.util.*;

public class ArrayListDemo {
   public static void main(String args[]) {
      // create an array list
      ArrayList al = new ArrayList();
      System.out.println("Initial size of al: " + al.size());

      // add elements to the array list
      al.add("C");
      al.add("A");
      al.add("E");
      al.add("B");
      al.add("D");
      al.add("F");
      al.add(1, "A2");
      System.out.println("Size of al after
                          additions: " + al.size());

      // display the array list
      System.out.println("Contents of al: " + al);
      // Remove elements from the array list
      al.remove("F");
      al.remove(2);
      System.out.println("Size of al after
                          deletions: " + al.size());
      System.out.println("Contents of al: " + al);
   }
}
```

Stack, the definition

- Stack is the composition of elements, for which the following operations are defined
  1. initialize (initializes/constructs the stack)
  2. push (push a new object to the top of the stack)
  3. pop (removes an object from the top of the stack)
  4. is_empty (test whether the stack is empty)
- When compared to the list
  - Ordering is always the order of arrival
  - Inserting only to one end (push)
  - Removing only from the beginning (pop)



Stack, application areas

- In principle, stack is a container, from which the elements can retrieved in reversed order (compared to the order of arrival)
- We can also say that the last element entered to the stack is retrieved first from the stack
  - For that reason we call stack as LIFO (Last In First Out list)
  - We can also say that the element inserted first to the stack is retrieved last from the stack (First In Last Out : FILO)
- In its simplest form we can use stack to reverse the ordering of elements
  - When we must reverse the ordering of characters in a string, we can do that by the stack
    - By pushing all the characters to the stack (of characters
    - After pushing, we pop the characters from the stack and insert them to the original string

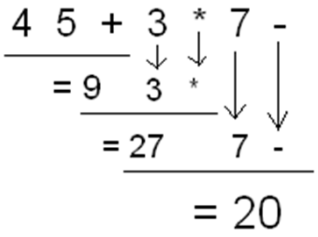
```
Stack<char> stack;
String string1;
Character c;

// read string and push every character to the stack
string1 = read_string();
for (int i = 0; i < string1.length(); i++)
    stack.push(string1.charAt(i));

// pop every character from the stack
// and insert them to the original string
String1 = "";
c = stack.pop();
while (c != null) {
    string1 = string1 + c;
    c = stack.pop();
}

print_string(string1);
```

Stack, application areas

- Reversing the ordering (e.g. reversing the string)
- Checking if a string is a palindrome (string that is the same independent of the reading order, e.g. radar, level, rotor, “Was it a rat I saw”)
  - String to be inspected is pushed to two different character stacks, stack 1 and 2
  - Then all the characters from stack 2 are pop’d and push’d to the new stack 3
  - The original string is a palindrome, if all characters pop’d character by character at the same rate from stacks 1 and 3) are same (until stacks are empty)
- Disassembling and assembling the equipment
  - We disassemble a complicated equipment, there may be difficulties to reassemble the equipment again. This can be helped with a stack
  - When we disassemble the equipment, we enter the id of the part removed. Our program will insert this id to the stack
  - When we assemble the equipment, we ask which part will be assembled this time. Our program takes the id from the stack.
  - With this program, we can be sure that the assembling is done exactly in reverse order (compared to disassembling)

Stack, application areas

- Calculation of Postfix equations
  - RPN calculators (Reverse Polish Notation) uses postfix notation
    - In that notation we mark the ”normal” equation (2+3)/5 as 2 3 + 5 /
    - Postfix notation is easier for the computer to check (than normal infix notation)
    - In postfix notation we can proceed incrementally from the left to the right, and the decision what to do is based only what we have encountered so far (there is no need for parenthesis)
    - Postscript language (used in printers) is based on postfix notation, e.g. C-language statement if (a<b) statement1; else statement2; can be written in postfix language as a b gt {statement1} {statement2} ifelse
    - Evaluation of the equation is based on the following simple rules:
      1. If we encounter operand, it will be pushed to the stack
      2. If we encounter operator, we take two values from the stack, process the operand for those values, and this new value will be pushed back to the stack
      3. When the whole equation has been processed, top of the stack will have the result of the evaluation

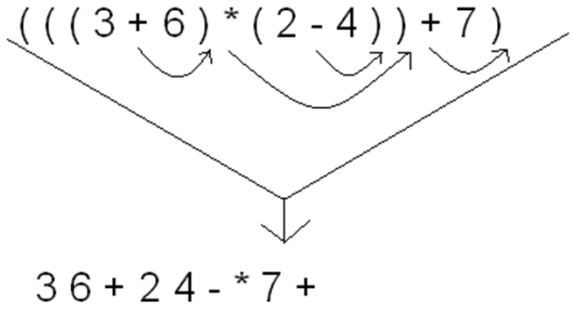



```
%!PS
/Courier
findfont
20 scalefont
setfont
72 500 moveto
(Hello world!)
show
showpage
```

HP 15C calculator which uses the RPN notation

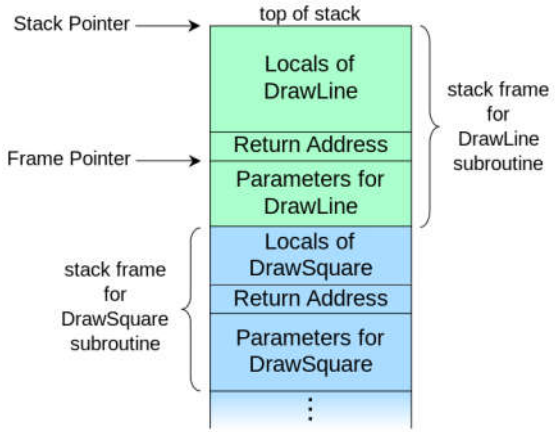
Stack, application areas

- Evaluation of normal infix equation is not very simple. Therefore implementing a program to evaluate infix equations with unlimited amount of parenthesis is a tough problem
  - In the case we don't know the stack container (and recursion)
- We previously noted, that postfix equations are very straightforward to evaluate
- Therefore the simplest way to evaluate infix equations is two phased:
  - First we transform infix equation to postfix form
  - Then we evaluate the postfix equation
- Transforming infix equation to postfix form
  1. If we find operand in infix equation, we put it to the end of the postfix equation
  2. If we find operator in infix equation, this will be pushed to the operator stack. Before that we remove all those operators which have precedence over the current operator. All the operators removed from the stack will be inserted to the end of postfix equation
  3. Finally we remove all operators from the stack and they will be inserted to the end of postfix equation
- Parenthesis are also considered as operators
- This algorithm is widely known as Shunting-yard algorithm by Edsger Dijkstra 1961



Stack, application areas

- Function calls inside functions and recursive function
  - Stack is used in computers to passing parameters to functions and reserving space for local variables
  - The benefit is the function can further call other function, and this function can call another function, etc.
    - When we return from the function, the function which made the call, recovers its original memory area from its local variables
      - Because we pop the memory area of the called function
  - Reserving parameters and local variables from the stack makes it possible to create recursive functions (functions that call themselves)
  - Interruptions (in program)
    - Interrupt return address is push'd to the stack
    - Other interrupt can interrupt the processing of the previous, return address will be automatically retrieved from the stack



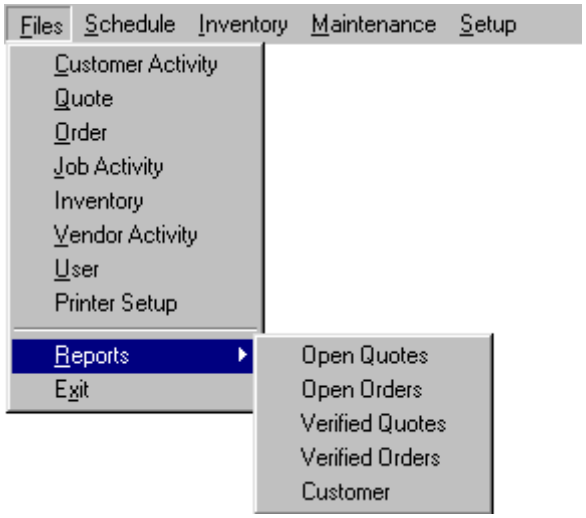

```
void DrawLine(…) {
    …
}

void DrawSquare(…) {
    …
    DrawLine(…);
    …
}
```

```
void f(int a) {
  if (a > 0)
    f(a-1);
  System.out.print(a + " ");
}
void main() {
  f(3);
}
```

We get the values 0, 1, 2 and 3

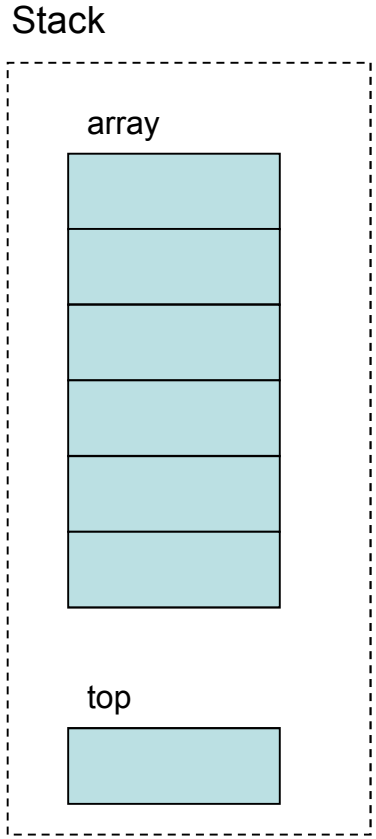
Stack, application areas

- Menu stack to manage nested menus
  - Window menu system (where we can select a new menu from the current menu, and further new menu from the selected new menu, etc.) can be implemented as menu stack
    - Then we can easily traverse back to the original menu
- Simple line editor
  - Using a stack it is very easy to implement simple line editor where we can delete characters from the end of the line with backspace-button
    - Entering a character means push operation, backspace means pop operation



Stack, array implementation

- One drawback of the list implementation was that inserting and removing elements (from the list) required movement of the elements in the list
- In the stack case, the ordering is always based on entering order, and removing the element is possible only from the end, we don't have the data movement problem in stack's implementation
  - Therefore the array implementation (of the stack) is very efficient, especially when the maximum size of the stack is known beforehand
- The idea of an array implementation is that the elements are stored in the array
- Top of the stack (variable top) announces the position in the array where lastly entered element resides
  - Top with value -1 means an empty stack




```
public class simpleStack<T extends Comparable<T>> {
    private static final int MAXN = 10;
    private int top;
    private T[] array;
```

Stack, array implementation

This is a container

```
public class simpleStack<T extends Comparable<T>> {
    private static final int MAXN = 10;
    private int top;
    private T[] array;

    public simpleStack() {
        top = -1;
        array = (T[])new Comparable[MAXN];
    }

    public boolean push(T item) {
        if (top >= MAXN-1)
            return false;
        else
            array[++top] = item;
            return true;
    }

    public T pop() {
        if (top == -1)
            return null;
        else
            return array[top--];
    }

    public void print() {
        for (int i = top; i >= 0; i--)
            System.out.print(array[i] + " ");
    }
```

```
public static void main(String[] args) {
        simpleStack<Character> stack = new simpleStack();
        Character             item;

        System.out.println("Enter a letter to push onto stack");
        System.out.println("or digit 1 to take a letter from stack");
        System.out.println("Return to end the program\n");
        try {
            item = new Character((char)System.in.read());
            while (item.compareTo(new Character('\n')) != 0) {
                if (item.compareTo(new Character('1')) == 0)
                    System.out.println("Letter popped from stack is " + stack.pop());
                else
                    stack.push(item);

                System.out.print("Stack content: "); stack.print(); System.out.println();
                item = new Character((char)System.in.read());
            }
            System.out.println();
        } catch(Exception e) {
            System.out.println("Exception " + e);
        }
    }
}
```