1. Stacks and Queues :: Stacks :: Introduction

A **stack** is one of the most widely used data structures in programming.

A stack is a linear (or sequential) data structure like a linked list, but unlike a linked list—where we can add, get, and remove elements from anywhere in the list—with a stack we are only permitted to add, get, and remove elements from one end of the linear sequence.

Since stack operations are only performed on one end of the linear sequence, the element on the end is referred to as the **top element**.

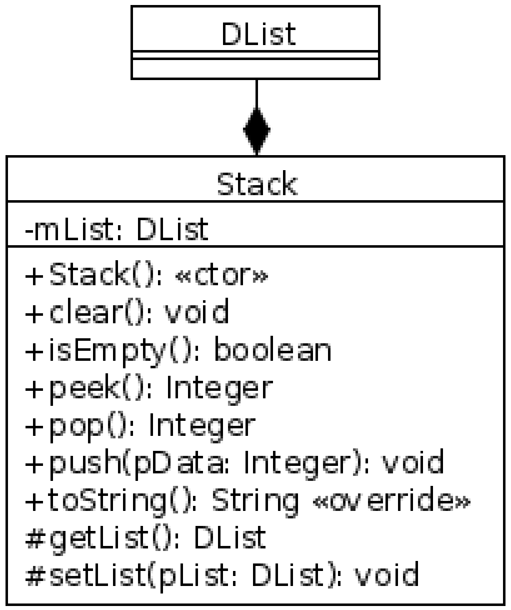
The standard stack operations are:

1. Push which adds a new element to the stack.
2. Peek which gets an element from the stack without removing it.
3. Pop which removes an element from the stack.

A stack is called a **last-in-first-out** (LIFO) data structure since the last element added to the structure is the first element removed.

1. Stacks and Queues :: Stacks :: UML Class Diagram

A stack can be easily implemented by using a linked list to store the elements of the stack. Our *Stack* class diagram is:



A *Stack* object contains one instance variable which is a *DList* (of *Integer*). The *clear*() method is called to remove all of the elements from the stack; after calling *clear*() the stack is empty and *isEmpty*() would return true.

getList() and *setList*() are protected accessor and mutator methods for the private *mList* instance variable. They are not intended to be publicly called on a *Stack* but are made protected so they may be called by subclasses of *Stack*.

1. Stacks and Queues :: Stacks :: Implementation

```
//***********************************************************************************
// CLASS: Stack (Stack.java)
//***********************************************************************************
// Implements a stack data structure using a DList to store the elements.
public class Stack {
    private DList mList;
    // Creates a new empty Stack by creating a new empty DList.
    public Stack() {
        setList(new DList());
    }
    // Removes all of the elements from this Stack. After clear() returns this Stack
    // is empty.
    public void clear() {
        getList().clear();
    }
    // Accessor method for mList.
    protected DList getList() {
        return mList;
    }
```

1. Stacks and Queues :: Stacks :: Implementation (continued)

```
// Returns true if this Stack is empty.
public boolean isEmpty() {
    return getList().isEmpty();
}

// Returns the top element on this Stack without removing it.
public Integer peek() {
    return getList().get(0);
}

// Removes the top element from this Stack and returns it.
public Integer pop() {
    Integer top = getList().remove(0);
    return top;
}

// Pushes pData onto the top of this Stack.
public void push(Integer pData) {
    getList().prepend(pData);
}

// Mutator method for mList.
protected void setList(DList pList) {
    mList = pList;
}
```

1. Stacks and Queues :: Stacks :: Implementation (continued)

```
    // Overrides toString() inherited from Object. Returns a String representation of
    // the elementsof this Stack by calling the DList.toString() method.
    @Override
    public String toString() {
        return getList().toString();
    }
}
```

1. Stacks and Queues :: Stacks :: Time Complexity

How efficient are the peek, push, and pop operations? From *Linked Lists* : *Section 11*:

1. **$\boldsymbol{add()}$, $\boldsymbol{get()}$, and $\boldsymbol{remove()}$ at the head and tail of a doubly linked list is $\boldsymbol{O(1)}$.**
2. $add()$, $get()$, and $remove()$ on an interior element—that is, randomly accessing elements—is $O(n)$.
3. Iterating over the elements of a linked list in sequence is an efficient operation as we can access the next element in the sequence in $O(1)$ time.

Therefore, a linked list is a good data structure for storing a linear sequence of elements where:

1. The number of elements in the list will grow and shrink, i.e., it is a dynamic data structure.
2. **Elements are primarily accessed at the beginning and ends of the list.**
3. Elements are accessed in a linear sequence.