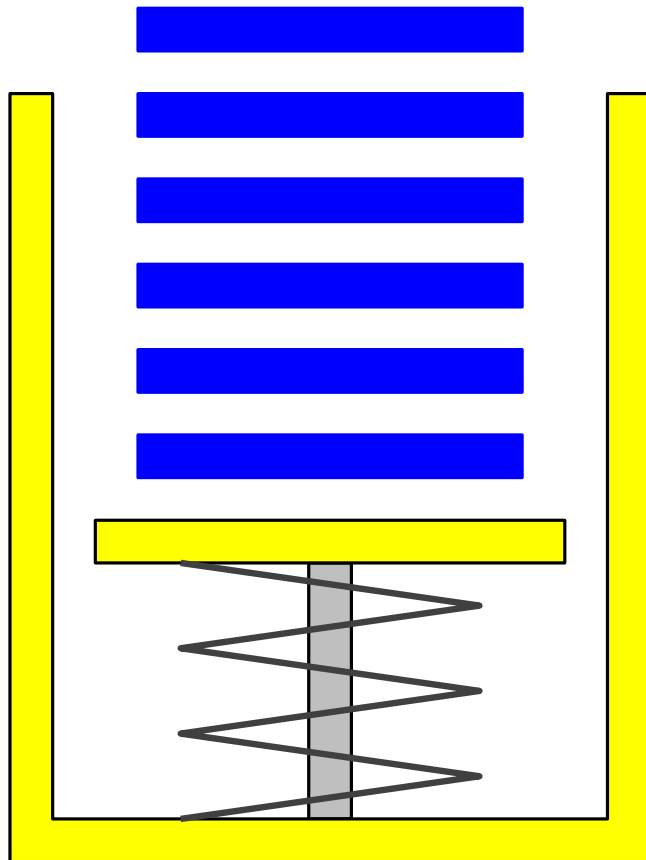


Stacks

Think about a stack of books, or dishes.

One can only access the top of the stack.



Operations on a stack:

- **push** something on top,
- look at the **top**,
- **pop** something off the stack.

The Stack interface

```
public interface Stack<AnyType> {  
    void clear();  
    boolean isEmpty();  
    AnyType top();  
    AnyType pop();  
    void push(AnyType x);  
}
```

This **interface** shows the specification of a stack as an **Abstract Data Type** (ADT).

An abstract data type is defined by

- what **operations** are allowed on it, and
- the **semantics** (meaning) of these operations.

Using a stack:

```
Stack<Integer> stack = new DsStack<Integer>();  
stack.push(5);  
int k = stack.top();  
stack.pop();
```

To create an object with interface `Stack`, we need to create a specific implementation of stacks. In this case we use `DsStack`.

Balanced symbol checker

$() \{ [() \{ }] ([]) \}$ is correct,
but $() \{ [(\{) \}] ([]) \}$ is not correct.

How to check whether a string is balanced:

1. Make an empty stack,
2. For each symbol in the string:
 - (a) If the symbol is an opening symbol, push it on the stack.
 - (b) If it is a closing symbol, then
 - i. If the stack is empty, return false.
 - ii. If the top of the stack does not match the closing symbol, return false.
 - iii. Pop the stack.
3. Return true if the stack is empty, otherwise false.

Stacks and recursion

The Java runtime system maintains **activation records** (stack frames) on a stack.

```
public static long factorial(int n)
{
    if (n <= 1)        // base case
        return 1;
    else
        return n * factorial(n - 1);
}
```

(This stack is built into the Java runtime system! It is not a Java object—we cannot access the stack of activation records ourselves.)