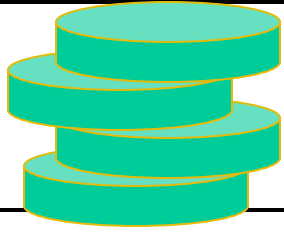
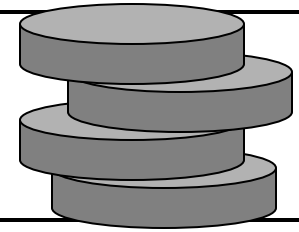

Stacks

Feb 18



Stacks



- Insertion and deletions are First In Last Out
 - FILO
 - or LIFO
- Physical stacks are everywhere!
- Function names (in the following slides) follow `java.util.Stack` rather than the book.

Stack Interface

- `null` is returned from `peek()` and `pop()` when stack is empty
 - throw exception?

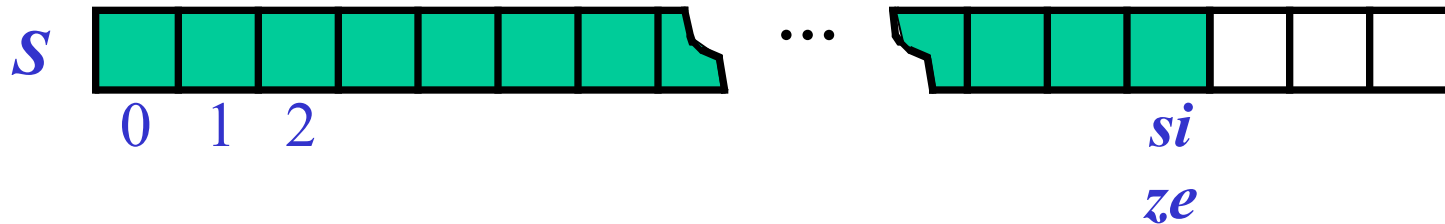
```
public interface StackInft<E> {  
    public boolean empty();  
    public E push(E e);  
    public E peek();  
    public E pop();  
    public int size();  
}
```

Example

Method	Return Value	Stack Contents
push(5)	5	{5}
push(3)	3	{5, 3}
size()	2	{5, 3}
pop()	3	{5}
empty()	FALSE	{5}
pop()	5	{}
empty()	TRUE	{}
pop()	null	{}
push(7)	7	{7}
push(9)	9	{7,9}
peek()	9	{7,9}

Array-based Stack

- Implement the stack with an array
- Add elements onto the end of the array
- Use an int `size` to keep track of the top



Performance and Limitations

- Performance

- let n be the number of objects in the stack
- The space used is $O(n)$
- Each operation runs in time $O(1)$

- Limitations

- Max size is limited and can not be changed
- Pushing onto a full stack can fail

Push

- Array has set size and may become full
- A `push` will fail if the array becomes full
 - Limitation of the array-based implementation
 - Alternatives?
 - Make the array grow (use `ArrayList`)?
 - Linked Lists?
 - What do to on fail?
 - return null
 - throw exception

Implementing an Array-based stack

```
public class ArrayStack<K> implements StackIntf<K> {
    private static final int DEFAULT_CAPACITY = 40;
    private int size;
    private K[] underlyingArray;

    public ArrayStack() {
        this(DEFAULT_CAPACITY);
    }

    public ArrayStack(int capacity) {
        size=0;
        underlyingArray = (K[]) new Object[capacity];
    }
}
```


Code

empty, peek and pop

About pop

```
public K pop() {  
    if (size>0) {  
        size--;  
        K tmp = underlyingArray[size];  
        underlyingArray[size]=null;  
        return tmp;  
    }  
    return null;  
}
```

Why?



Method Stack in the JVM

- The JVM keeps track of the chain of active methods with a stack
 - `printStackTrace()` — only within catch block of exception
 - `Thread.dumpStack()` — anywhere
- On a method call, the JVM pushes onto the stack a frame containing:
 - parameters
 - local variables
 - return address
- When a method ends, control passes onto the method on top of the stack
- Using VSC to view the stack — `MethodStack.java`

Stack Applications

- Reversing
- Prefix/postfix algebraic interpreter
- Palindromes
 - Madam Im adam
 - A man a plan a canal panama!
 - Dennis, Nell, Edna, Leon, Nedra, Anita, Rolf, Nora, Alice, Carol, Leo, Jane, Reed, Dena, Dale, Basil, Rae, Penny, Lana, Dave, Denny, Lena, Ida, Bernadette, Ben, Ray, Lila, Nina, Jo, Ira, Mara, Sara, Mario, Jan, Ina, Lily, Arne, Bette, Dan, Reba, Diane, Lynn, Ed, Eva, Dana, Lynne, Pearl, Isabel, Ada, Ned, Dee, Rena, Joel, Lora, Cecil, Aaron, Flora, Tina, Arden, Noel, and Ellen sinned.
- Recursion
- OS Tasks

Using Stacks

Method	Return Value	Stack Contents
push(5)	5	{5}
push(3)	3	{5, 3}
size()	2	{5, 3}
pop()	3	{5}
empty()	FALSE	{5}
pop()	5	{}
empty()	TRUE	{}
pop()	null	{}
push(7)	7	{7}
push(9)	9	{7,9}
peek()	9	{7,9}