

Stacks

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2019



Outline

Stack



Stack



- ▶ A **Stack** is a standard Interface

Stack



- ▶ A **Stack** is a standard Interface
 - ▶ which is so standard

Stack



- ▶ A **Stack** is a standard Interface
 - ▶ which is so standard
 - ▶ that Java didn't even bother making it an Interface.

Stack



- ▶ A **Stack** is a standard Interface
 - ▶ which is so standard
 - ▶ that Java didn't even bother making it an Interface.
- ▶ Like any kind of stack we can think of,

Stack



- ▶ A **Stack** is a standard Interface
 - ▶ which is so standard
 - ▶ that Java didn't even bother making it an Interface.
- ▶ Like any kind of stack we can think of,
 - ▶ the top entry is easy to add, view, or remove.

Stack



- ▶ A **Stack** is a standard Interface
 - ▶ which is so standard
 - ▶ that Java didn't even bother making it an Interface.
- ▶ Like any kind of stack we can think of,
 - ▶ the top entry is easy to add, view, or remove.
 - ▶ Trying to add, view, or remove entries in the middle is messy and awkward.

Stack Methods

Stack Methods

- ▶ The names for the Stack methods are a little strange:



Stack Methods

- ▶ The names for the Stack methods are a little strange:
 - ▶ **push** add a new entry to the top of the stack



Stack Methods

- ▶ The names for the Stack methods are a little strange:
 - ▶ **push** add a new entry to the top of the stack
 - ▶ **pop** remove one entry from the top of the stack



Stack Methods

- ▶ The names for the Stack methods are a little strange:
 - ▶ **push** add a new entry to the top of the stack
 - ▶ **pop** remove one entry from the top of the stack
 - ▶ **peek** look at the top entry of the stack without changing it



Stack Methods

- ▶ The names for the Stack methods are a little strange:
 - ▶ **push** add a new entry to the top of the stack
 - ▶ **pop** remove one entry from the top of the stack
 - ▶ **peek** look at the top entry of the stack without changing it
 - ▶ **empty** true if there is nothing in the stack, false otherwise



Stack Methods

- ▶ The names for the Stack methods are a little strange:
 - ▶ **push** add a new entry to the top of the stack
 - ▶ **pop** remove one entry from the top of the stack
 - ▶ **peek** look at the top entry of the stack without changing it
 - ▶ **empty** true if there is nothing in the stack, false otherwise
- ▶ When I put something on top of one of the towering stacks of papers on my desk,



Stack Methods

- ▶ The names for the Stack methods are a little strange:
 - ▶ **push** add a new entry to the top of the stack
 - ▶ **pop** remove one entry from the top of the stack
 - ▶ **peek** look at the top entry of the stack without changing it
 - ▶ **empty** true if there is nothing in the stack, false otherwise
- ▶ When I put something on top of one of the towering stacks of papers on my desk,
 - ▶ I don't think of it as *pushing*,



Stack Methods

- ▶ The names for the Stack methods are a little strange:
 - ▶ **push** add a new entry to the top of the stack
 - ▶ **pop** remove one entry from the top of the stack
 - ▶ **peek** look at the top entry of the stack without changing it
 - ▶ **empty** true if there is nothing in the stack, false otherwise
- ▶ When I put something on top of one of the towering stacks of papers on my desk,
 - ▶ I don't think of it as *pushing*,
 - ▶ nor do I think of it as *popping* when I remove it.



Stack Methods

- ▶ The names for the Stack methods are a little strange:
 - ▶ **push** add a new entry to the top of the stack
 - ▶ **pop** remove one entry from the top of the stack
 - ▶ **peek** look at the top entry of the stack without changing it
 - ▶ **empty** true if there is nothing in the stack, false otherwise
- ▶ When I put something on top of one of the towering stacks of papers on my desk,
 - ▶ I don't think of it as *pushing*,
 - ▶ nor do I think of it as *popping* when I remove it.
 - ▶ Peek and empty make sense though.



Name Origins

I think what the original inventors had in mind was a 1950s buffet diner spring loaded plate dispenser.



Name Origins

I think what the original inventors had in mind was a 1950s buffet diner spring loaded plate dispenser.



- ▶ The power cord is to run a dish warmer.

Name Origins

I think what the original inventors had in mind was a 1950s buffet diner spring loaded plate dispenser.



- ▶ The power cord is to run a dish warmer.
- ▶ It doesn't shoot the dishes up when it pops!

Name Origins

I think what the original inventors had in mind was a 1950s buffet diner spring loaded plate dispenser.



- ▶ The power cord is to run a dish warmer.
- ▶ It doesn't shoot the dishes up when it pops!
- ▶ Instead, it always keeps the top dish level with the top of the dispenser,

Name Origins

I think what the original inventors had in mind was a 1950s buffet diner spring loaded plate dispenser.



- ▶ The power cord is to run a dish warmer.
- ▶ It doesn't shoot the dishes up when it pops!
- ▶ Instead, it always keeps the top dish level with the top of the dispenser,
- ▶ although I don't think that requires electricity.



Stack methods in action

```
Stack stack = new Stack();
```

Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");  
stack.push("coconut");
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");  
stack.push("coconut");  
stack.pop();             // returns "coconut"
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");  
stack.push("coconut");  
stack.pop();             // returns "coconut"  
stack.peek();            // returns "banana"
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");  
stack.push("coconut");  
stack.pop();             // returns "coconut"  
stack.peek();            // returns "banana"  
stack.push("cantaloupe");
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");  
stack.push("coconut");  
stack.pop();             // returns "coconut"  
stack.peek();            // returns "banana"  
stack.push("cantaloupe");  
stack.pop();             // returns "cantaloupe"
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");  
stack.push("coconut");  
stack.pop();             // returns "coconut"  
stack.peek();            // returns "banana"  
stack.push("cantaloupe");  
stack.pop();             // returns "cantaloupe"  
stack.pop();             // returns "banana"
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");  
stack.push("coconut");  
stack.pop();             // returns "coconut"  
stack.peek();            // returns "banana"  
stack.push("cantaloupe");  
stack.pop();             // returns "cantaloupe"  
stack.pop();             // returns "banana"  
stack.empty();           // returns false
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");  
stack.push("coconut");  
stack.pop();             // returns "coconut"  
stack.peek();            // returns "banana"  
stack.push("cantaloupe");  
stack.pop();             // returns "cantaloupe"  
stack.pop();             // returns "banana"  
stack.empty();           // returns false  
stack.pop();             // returns "mango"
```



Stack methods in action

```
Stack stack = new Stack();  
stack.empty();           // returns true  
stack.push("mango");  
stack.push("banana");  
stack.push("coconut");  
stack.pop();             // returns "coconut"  
stack.peek();            // returns "banana"  
stack.push("cantaloupe");  
stack.pop();             // returns "cantaloupe"  
stack.pop();             // returns "banana"  
stack.empty();           // returns false  
stack.pop();             // returns "mango"  
stack.peek();            // throws Exception
```



Lab

For the next lab, you will learn three ways to implement Stack.



Lab

For the next lab, you will learn three ways to implement Stack.
In `StackInt.java`, you will notice something new:



Lab

For the next lab, you will learn three ways to implement Stack.
In `StackInt.java`, you will notice something new:

▶ < E >



Lab

For the next lab, you will learn three ways to implement Stack.
In `StackInt.java`, you will notice something new:

- ▶ `< E >`

That is a generic declaration. In means you can have



Lab

For the next lab, you will learn three ways to implement Stack.
In `StackInt.java`, you will notice something new:

- ▶ `< E >`

That is a generic declaration. In means you can have

- ▶ `StackInt<String>`



Lab

For the next lab, you will learn three ways to implement Stack. In `StackInt.java`, you will notice something new:

- ▶ `< E >`

That is a generic declaration. In means you can have

- ▶ `StackInt<String>`
- ▶ `StackInt<DirectoryEntry>`



Lab

For the next lab, you will learn three ways to implement Stack. In `StackInt.java`, you will notice something new:

- ▶ `< E >`

That is a generic declaration. In means you can have

- ▶ `StackInt<String>`
- ▶ `StackInt<DirectoryEntry>`
- ▶ or a stack of any type of class.



Lab

For the next lab, you will learn three ways to implement Stack. In `StackInt.java`, you will notice something new:

- ▶ `< E >`

That is a generic declaration. In means you can have

- ▶ `StackInt<String>`
- ▶ `StackInt<DirectoryEntry>`
- ▶ or a stack of any type of class.

When you do this, the Java compiler will make sure you only put that kind of thing into that stack.



Lab

For the next lab, you will learn three ways to implement Stack. In `StackInt.java`, you will notice something new:

- ▶ `< E >`

That is a generic declaration. In means you can have

- ▶ `StackInt<String>`
- ▶ `StackInt<DirectoryEntry>`
- ▶ or a stack of any type of class.

When you do this, the Java compiler will make sure you only put that kind of thing into that stack.

It has to be a class, however, so for primitive data types you have to use the class version of those types:



Lab

For the next lab, you will learn three ways to implement Stack.

In **StackInt.java**, you will notice something new:

- ▶ `< E >`

That is a generic declaration. In means you can have

- ▶ `StackInt<String>`
- ▶ `StackInt<DirectoryEntry>`
- ▶ or a stack of any type of class.

When you do this, the Java compiler will make sure you only put that kind of thing into that stack.

It has to be a class, however, so for primitive data types you have to use the class version of those types:

- ▶ `char` → `Character`



Lab

For the next lab, you will learn three ways to implement Stack. In **StackInt.java**, you will notice something new:

- ▶ `< E >`

That is a generic declaration. It means you can have

- ▶ `StackInt<String>`
- ▶ `StackInt<DirectoryEntry>`
- ▶ or a stack of any type of class.

When you do this, the Java compiler will make sure you only put that kind of thing into that stack.

It has to be a class, however, so for primitive data types you have to use the class version of those types:

- ▶ `char` → `Character`
- ▶ `int` → `Integer`



Lab

For the next lab, you will learn three ways to implement Stack. In `StackInt.java`, you will notice something new:

- ▶ `< E >`

That is a generic declaration. It means you can have

- ▶ `StackInt<String>`
- ▶ `StackInt<DirectoryEntry>`
- ▶ or a stack of any type of class.

When you do this, the Java compiler will make sure you only put that kind of thing into that stack.

It has to be a class, however, so for primitive data types you have to use the class version of those types:

- ▶ `char` → `Character`
- ▶ `int` → `Integer`
- ▶ `double` → `Double`



For the next lab, you will learn three ways to implement Stack.

In `StackInt.java`, you will notice something new:

- ▶ `< E >`

That is a generic declaration. It means you can have

- ▶ `StackInt<String>`
- ▶ `StackInt<DirectoryEntry>`
- ▶ or a stack of any type of class.

When you do this, the Java compiler will make sure you only put that kind of thing into that stack.

It has to be a class, however, so for primitive data types you have to use the class version of those types:

- ▶ `char` → `Character`
- ▶ `int` → `Integer`
- ▶ `double` → `Double`

This is less efficient (by a constant factor in space and time) than creating a specific `StackOfChar`, etc., but it is usually good enough.

ArrayStack

ArrayStack.java



ArrayStack

ArrayStack.java

- ▶ Array based implementation of StackInt.



ArrayStack

ArrayStack.java

- ▶ Array based implementation of StackInt.
- ▶ Entries are pushed at the end (max index) of the array.



ArrayStack

ArrayStack.java

- ▶ Array based implementation of StackInt.
- ▶ Entries are pushed at the end (max index) of the array.
- ▶ So push is $O(1)$,



ArrayStack

ArrayStack.java

- ▶ Array based implementation of StackInt.
- ▶ Entries are pushed at the end (max index) of the array.
- ▶ So push is $O(1)$,
- ▶ (unless the array is full and needs to be reallocated).



ArrayStack.java

- ▶ Array based implementation of StackInt.
- ▶ Entries are pushed at the end (max index) of the array.
- ▶ So push is $O(1)$,
- ▶ (unless the array is full and needs to be reallocated).
- ▶ This is the fastest way to implement a stack,

ArrayStack.java

- ▶ Array based implementation of StackInt.
- ▶ Entries are pushed at the end (max index) of the array.
- ▶ So push is $O(1)$,
- ▶ (unless the array is full and needs to be reallocated).
- ▶ This is the fastest way to implement a stack,
- ▶ but it might not be good for real time programming.

(Sorry the laser stopped in the middle of your eye, but we have to allocate a bigger array!)

LinkedStack

LinkedStack.java

LinkedStack

LinkedStack.java

- ▶ Linked list implementation



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).
- ▶ `data.next` gets you the next entry instead of `data.getNext()`.



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).
- ▶ `data.next` gets you the next entry instead of `data.getNext()`.

Other changes:



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).
- ▶ `data.next` gets you the next entry instead of `data.getNext()`.

Other changes:

- ▶ The Node is singly linked instead of doubly linked.



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).
- ▶ `data.next` gets you the next entry instead of `data.getNext()`.

Other changes:

- ▶ The Node is singly linked instead of doubly linked.
- ▶ There is no previous.



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).
- ▶ `data.next` gets you the next entry instead of `data.getNext()`.

Other changes:

- ▶ The Node is singly linked instead of doubly linked.
- ▶ There is no previous.
- ▶ Saves space and time.



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).
- ▶ `data.next` gets you the next entry instead of `data.getNext()`.

Other changes:

- ▶ The Node is singly linked instead of doubly linked.
- ▶ There is no previous.
- ▶ Saves space and time.
- ▶ Works fine for this specialized application.



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).
- ▶ `data.next` gets you the next entry instead of `data.getNext()`.

Other changes:

- ▶ The Node is singly linked instead of doubly linked.
- ▶ There is no previous.
- ▶ Saves space and time.
- ▶ Works fine for this specialized application.

As a result:



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).
- ▶ `data.next` gets you the next entry instead of `data.getNext()`.

Other changes:

- ▶ The Node is singly linked instead of doubly linked.
- ▶ There is no previous.
- ▶ Saves space and time.
- ▶ Works fine for this specialized application.

As a result:

- ▶ Pushing and popping are done at the *beginning* of the list,



LinkedList

LinkedList.java

- ▶ Linked list implementation
- ▶ $O(1)$ per operation (really?).

You will notice some new techniques.

- ▶ The entire Node class is private
- ▶ and declared inside LinkedList.
- ▶ No separate Java file
- ▶ No need for accessor methods (getNext(), etc).
- ▶ `data.next` gets you the next entry instead of `data.getNext()`.

Other changes:

- ▶ The Node is singly linked instead of doubly linked.
- ▶ There is no previous.
- ▶ Saves space and time.
- ▶ Works fine for this specialized application.

As a result:

- ▶ Pushing and popping are done at the *beginning* of the list,
- ▶ *not* the end.



ListStack

ListStack.java



ListStack

ListStack.java

- ▶ Implementation using `java.util.List`



ListStack

ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.



ListStack

ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.

List is an *interface*



ListStack

ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.

List is an *interface*

- ▶ Describes a list.



ListStack

ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.

List is an *interface*

- ▶ Describes a list.
- ▶ `add(item)` means add an item to the end of the list.



ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.

List is an *interface*

- ▶ Describes a list.
- ▶ `add(item)` means add an item to the end of the list.
- ▶ We will use `add()` to implement `push()`.

ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.

List is an *interface*

- ▶ Describes a list.
- ▶ `add(item)` means add an item to the end of the list.
- ▶ We will use `add()` to implement `push()`.

Look at the List documentation,

ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.

List is an *interface*

- ▶ Describes a list.
- ▶ `add(item)` means add an item to the end of the list.
- ▶ We will use `add()` to implement `push()`.

Look at the List documentation,

- ▶ particularly `size()`, `get()`, and `remove()`.

ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.

List is an *interface*

- ▶ Describes a list.
- ▶ `add(item)` means add an item to the end of the list.
- ▶ We will use `add()` to implement `push()`.

Look at the List documentation,

- ▶ particularly `size()`, `get()`, and `remove()`.
- ▶ How do we implement `empty()`?

ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.

List is an *interface*

- ▶ Describes a list.
- ▶ `add(item)` means add an item to the end of the list.
- ▶ We will use `add()` to implement `push()`.

Look at the List documentation,

- ▶ particularly `size()`, `get()`, and `remove()`.
- ▶ How do we implement `empty()`?
- ▶ How do we implement `peek()`?

ListStack.java

- ▶ Implementation using `java.util.List`
- ▶ and its implementation `java.util.ArrayList`.

List is an *interface*

- ▶ Describes a list.
- ▶ `add(item)` means add an item to the end of the list.
- ▶ We will use `add()` to implement `push()`.

Look at the List documentation,

- ▶ particularly `size()`, `get()`, and `remove()`.
- ▶ How do we implement `empty()`?
- ▶ How do we implement `peek()`?
- ▶ How do we implement `pop()`?

Use ArrayList implementation of List.

Use ArrayList implementation of List.

- ▶ Partially filled array.

Use ArrayList implementation of List.

- ▶ Partially filled array.
- ▶ Just like we have been doing.

Use ArrayList implementation of List.

- ▶ Partially filled array.
- ▶ Just like we have been doing.
- ▶ When `size==length`, it reallocates.

Use ArrayList implementation of List.

- ▶ Partially filled array.
- ▶ Just like we have been doing.
- ▶ When `size==length`, it reallocates.
- ▶ Array variable and size are private.

Use ArrayList implementation of List.

- ▶ Partially filled array.
- ▶ Just like we have been doing.
- ▶ When `size==length`, it reallocates.
- ▶ Array variable and size are private.

`java.util.LinkedList`

Use ArrayList implementation of List.

- ▶ Partially filled array.
- ▶ Just like we have been doing.
- ▶ When `size==length`, it reallocates.
- ▶ Array variable and size are private.

`java.util.LinkedList`

- ▶ Doubly linked list implementation of List.

Use ArrayList implementation of List.

- ▶ Partially filled array.
- ▶ Just like we have been doing.
- ▶ When `size==length`, it reallocates.
- ▶ Array variable and size are private.

`java.util.LinkedList`

- ▶ Doubly linked list implementation of List.
- ▶ We could easily use it if we wanted to,

Use ArrayList implementation of List.

- ▶ Partially filled array.
- ▶ Just like we have been doing.
- ▶ When `size==length`, it reallocates.
- ▶ Array variable and size are private.

`java.util.LinkedList`

- ▶ Doubly linked list implementation of List.
- ▶ We could easily use it if we wanted to,
- ▶ thanks to the List interface.

Summary



Summary

Stack



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `realloc()`.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `realloc()`.

LinkedStack



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `realloc()`.

LinkedStack

- ▶ Private Node class.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `realloc()`.

LinkedStack

- ▶ Private Node class.
- ▶ `node.next` instead of `node.getNext()`



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `reallocate()`.

LinkedStack

- ▶ Private Node class.
- ▶ `node.next` instead of `node.getNext()`
- ▶ Push and pop at front (head) of list.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `reallocate()`.

LinkedStack

- ▶ Private Node class.
- ▶ `node.next` instead of `node.getNext()`
- ▶ Push and pop at front (head) of list.

ListStack



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `reallocate()`.

LinkedStack

- ▶ Private Node class.
- ▶ `node.next` instead of `node.getNext()`
- ▶ Push and pop at front (head) of list.

ListStack

- ▶ Use Java *List* interface.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `reallocate()`.

LinkedStack

- ▶ Private Node class.
- ▶ `node.next` instead of `node.getNext()`
- ▶ Push and pop at front (head) of list.

ListStack

- ▶ Use Java *List* interface.
- ▶ Use `add(item)`, `size()`, `get(index)`, `remove(index)`.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `reallocate()`.

LinkedStack

- ▶ Private Node class.
- ▶ `node.next` instead of `node.getNext()`
- ▶ Push and pop at front (head) of list.

ListStack

- ▶ Use Java *List* interface.
- ▶ Use `add(item)`, `size()`, `get(index)`, `remove(index)`.
- ▶ ArrayList implementation uses partially filled array.



Summary

Stack

- ▶ The StackInt interface describes a *Stack*.
- ▶ Only adding or removing at the top is possible.
- ▶ Operations called *push*, *pop*, *peek*, *empty*.
- ▶ Implemented using array, linked list, or List interface.

ArrayStack

- ▶ Implement using an array.
- ▶ Adding is $O(1)$ except for `realloc()`.

LinkedList

- ▶ Private Node class.
- ▶ `node.next` instead of `node.getNext()`
- ▶ Push and pop at front (head) of list.

ListStack

- ▶ Use Java *List* interface.
- ▶ Use `add(item)`, `size()`, `get(index)`, `remove(index)`.
- ▶ ArrayList implementation uses partially filled array.
- ▶ LinkedList is another implementation of List using a doubly linked list.

