# Stacks

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2024

# 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();
```

# 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();
```

# 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();
```

# 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();
```

# 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();
```

# 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();
```

# 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();
```

# 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();
```

# 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 EmptyStackException
```

# Lab

# Lab

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

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- < E >

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- < E >

That is a generic declaration. In means you can have

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- < E >

That is a generic declaration. In means you can have

- StackInterface<String>

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- < E >

That is a generic declaration. In means you can have

- StackInterface<String>
- StackInterface<DirectoryEntry>

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- < E >

That is a generic declaration. In means you can have

- StackInterface<String>
- StackInterface<DirectoryEntry>
- or a stack of any type of class.

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- < E >

That is a generic declaration. In means you can have

- StackInterface<String>
- StackInterface<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 prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- < E >

That is a generic declaration. In means you can have

- StackInterface<String>
- StackInterface<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 prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- $< E >$

That is a generic declaration. In means you can have

- StackInterface<String>
- StackInterface<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 $\rightarrow$ Character

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- $<$ E $>$

That is a generic declaration. In means you can have

- StackInterface$<$String$>$
- StackInterface$<$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 $\rightarrow$ Character
- int $\rightarrow$ Integer

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- $<$ E $>$

That is a generic declaration. In means you can have

- StackInterface<String>
- StackInterface<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 $\rightarrow$ Character
- int $\rightarrow$ Integer
- double $\rightarrow$ Double

# Lab

For the next prog, you will learn three ways to implement Stack.
In StackInterface.java, you will notice something new:

- $<$ E $>$

That is a generic declaration. In means you can have

- StackInterface<String>
- StackInterface<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 $\rightarrow$ Character
- int $\rightarrow$ Integer
- double $\rightarrow$ 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.

# Examples

Stack<Puppy>

Stack<Cat>

Stack<Stack<Cash>>

# ArrayStack

ArrayStack.java

# ArrayStack

ArrayStack.java

- Array based implementation of StackInterface.

# ArrayStack

ArrayStack.java

- Array based implementation of StackInterface.
- Items are pushed at first unused location of the array.

# ArrayStack

ArrayStack.java

- Array based implementation of StackInterface.
- Items are pushed at first unused location of the array.
- So push is O(1),

# ArrayStack

ArrayStack.java

- Array based implementation of StackInterface.
- Items are pushed at first unused location of the array.
- So push is O(1),
- (unless the array is full and needs to be reallocated).

# ArrayStack

ArrayStack.java

- Array based implementation of StackInterface.
- Items are pushed at first unused location 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

ArrayStack.java

- Array based implementation of StackInterface.
- Items are pushed at first unused location 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

# LinkedStack

## LinkedStack.java

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

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

A linked list is like a scavenger hunt:

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

A linked list is like a scavenger hunt:

- top points to the Node containing the top data item.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

A linked list is like a scavenger hunt:

- top points to the Node containing the top data item.
- The next field of that Node, accessed by top.next,

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

A linked list is like a scavenger hunt:

- top points to the Node containing the top data item.
- The next field of that Node, accessed by top.next,
  points to the Node containing the next data item (down) in the stack.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

A linked list is like a scavenger hunt:

- top points to the Node containing the top data item.
- The next field of that Node, accessed by top.next,
  points to the Node containing the next data item (down) in the stack.
- The next of that Node points to the Node with the next data item

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

A linked list is like a scavenger hunt:

- top points to the Node containing the top data item.
- The next field of that Node, accessed by top.next,
  points to the Node containing the next data item (down) in the stack.
- The next of that Node points to the Node with the next data item
- and so forth.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

A linked list is like a scavenger hunt:

- top points to the Node containing the top data item.
- The next field of that Node, accessed by top.next,
  points to the Node containing the next data item (down) in the stack.
- The next of that Node points to the Node with the next data item
- and so forth.

To push:

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

A linked list is like a scavenger hunt:

- top points to the Node containing the top data item.
- The next field of that Node, accessed by top.next,
  points to the Node containing the next data item (down) in the stack.
- The next of that Node points to the Node with the next data item
- and so forth.

To push:

- Set newNode to a new Node with the new data item.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

A linked list is like a scavenger hunt:

- top points to the Node containing the top data item.
- The next field of that Node, accessed by top.next,
  points to the Node containing the next data item (down) in the stack.
- The next of that Node points to the Node with the next data item
- and so forth.

To push:

- Set newNode to a new Node with the new data item.
- Set its next to the current top Node.

# LinkedStack

## LinkedStack.java

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

You will notice some new techniques.

- The entire Node class is private and declared inside LinkedStack.
- No separate Java file (like DirectoryEntry.java).
- No need for accessor methods (getData(), etc).
- node.data gives you the data.
- I made it final since you are not supposed to change it.
- node.next gets you the next node.

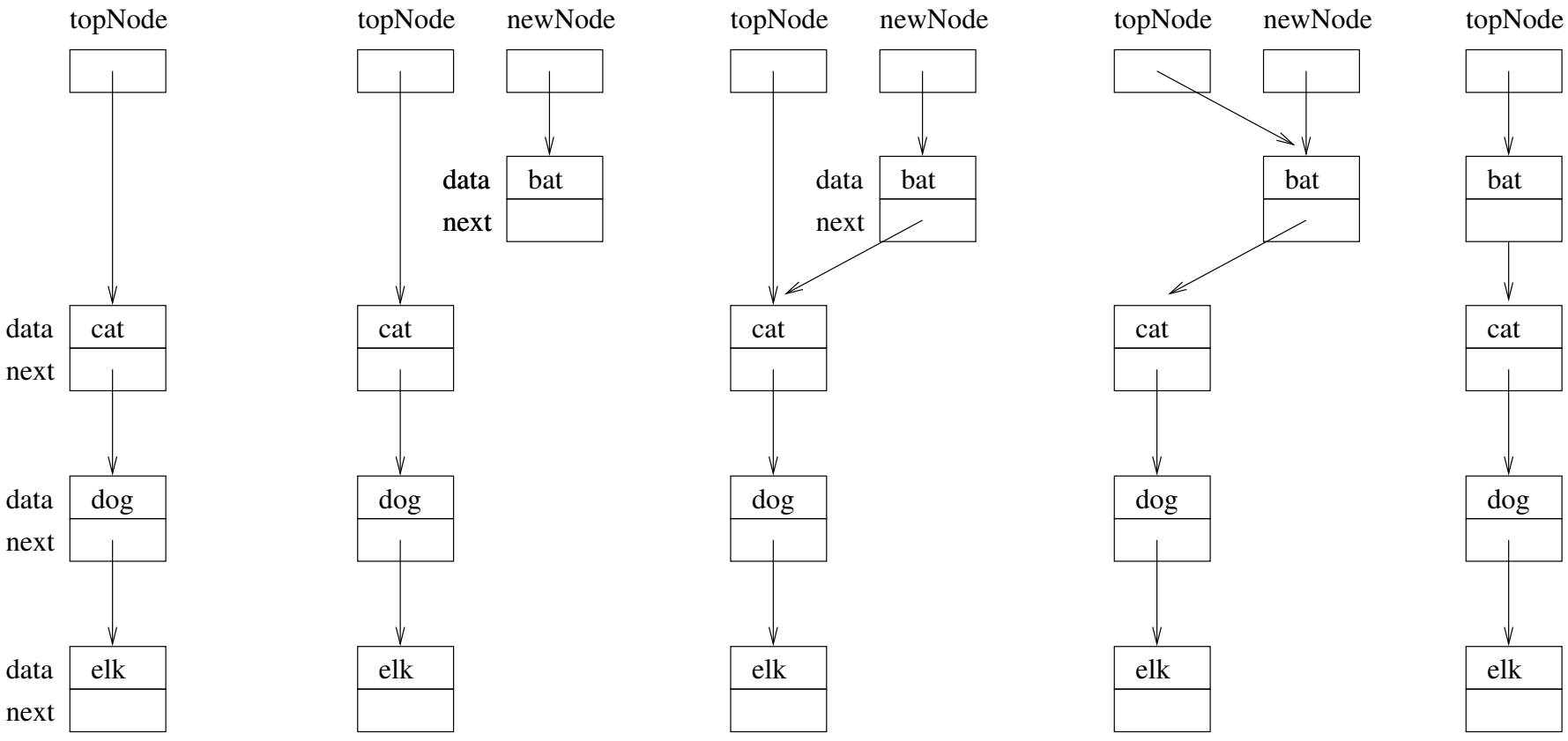
A linked list is like a scavenger hunt:

- top points to the Node containing the top data item.
- The next field of that Node, accessed by top.next,
  points to the Node containing the next data item (down) in the stack.
- The next of that Node points to the Node with the next data item
- and so forth.

To push:

- Set newNode to a new Node with the new data item.
- Set its next to the current top Node.
- Set top to the new Node.

# LinkedStack



# ListStack

# 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

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

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

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

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

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

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()?

# ListStack

Use ArrayList implementation of List.

# ListStack

Use ArrayList implementation of List.

- Partially filled array.

# ListStack

Use ArrayList implementation of List.

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

# ListStack

Use ArrayList implementation of List.

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

# ListStack

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.

# ListStack

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

# ListStack

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.

# ListStack

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.
- “Doubly” means each Node has a next and a previous.

# ListStack

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.
- “Doubly” means each Node has a next and a previous.
- We could easily use it if we wanted to,

# ListStack

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.
- “Doubly” means each Node has a next and a previous.
- We could easily use it if we wanted to,
- thanks to the List interface.

# Summary

# Summary

- Stack

# Summary

## Stack

- The StackInterface interface describes a *Stack*.

# Summary

## Stack

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

# Summary

## Stack

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

# Summary

## Stack

- The StackInterface 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 StackInterface 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 StackInterface 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 StackInterface 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 $\mathrm{O}(1)$ except for reallocate().

# Summary

Stack

- The StackInterface 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 $\mathrm{O}(1)$ except for reallocate().

LinkedStack

# Summary

Stack

- The StackInterface 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 $\mathrm{O}(1)$ except for reallocate().

LinkedStack

- Private Node class.

# Summary

Stack

- The StackInterface 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.data and node.next instead of node.getData() and node.getNext()

# Summary

Stack

- The StackInterface 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.data and node.next instead of node.getData() and node.getNext()
- Push and pop at front (top) of list.

# Summary

Stack

- The StackInterface 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 $\mathrm{O}(1)$ except for reallocate().

LinkedStack

- Private Node class.
- node.data and node.next instead of node.getData() and node.getNext()
- Push and pop at front (top) of list.

ListStack

# Summary

Stack

- The StackInterface 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.data and node.next instead of node.getData() and node.getNext()
- Push and pop at front (top) of list.

ListStack

- Use Java *List* interface.

# Summary

Stack

- The StackInterface 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.data and node.next instead of node.getData() and node.getNext()
- Push and pop at front (top) of list.

ListStack

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

# Summary

Stack

- The StackInterface 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.data and node.next instead of node.getData() and node.getNext()
- Push and pop at front (top) of list.

ListStack

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

# Summary

Stack

- The StackInterface 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.data and node.next instead of node.getData() and node.getNext()
- Push and pop at front (top) 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.