+

Java Collections Overview & Stack Applications

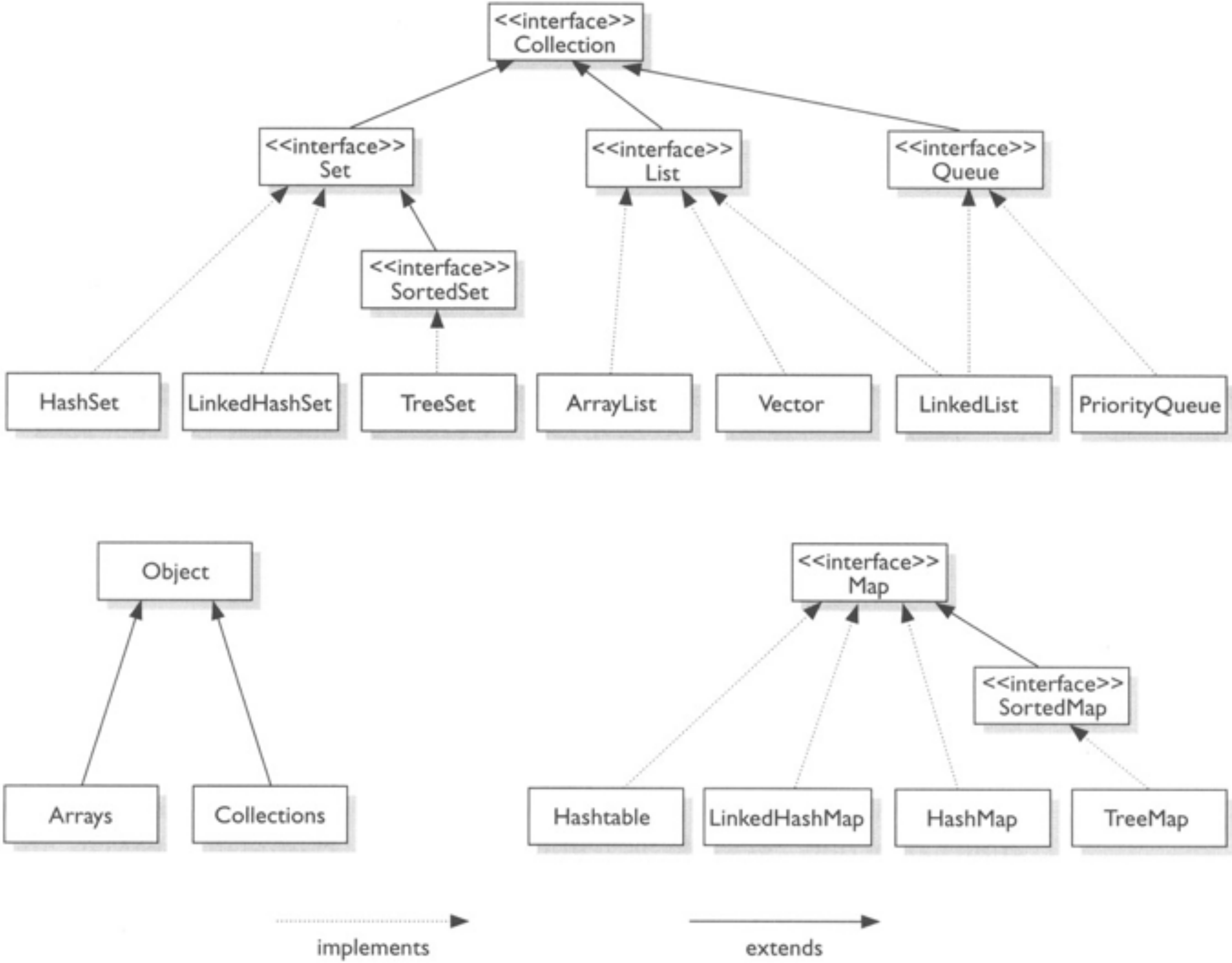
Questions?

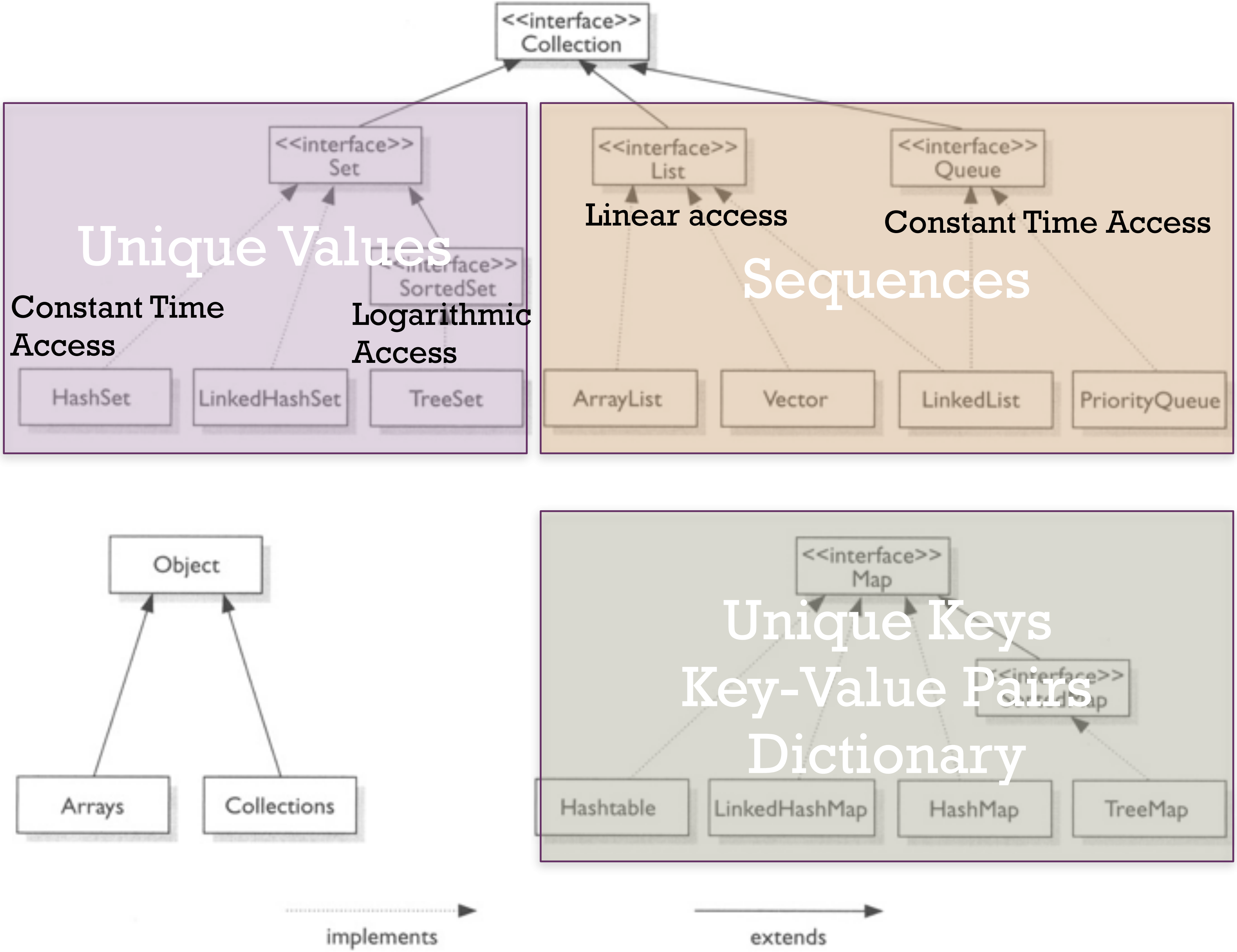
Assignment 4

- when looping through a list
  - consider using an iterator
  - if you call get(i) then only call it once and store it if you need it again.
  - make sure that you loop through all elements
    - for(int i = 0, i < list.size(); i++), YES
    - for(int i = 1, i < list.size(); i++), NO
    - for(int i = 0, i < list.size() - 1; i++), NO
    - for(int i = 0, i < list.size(); i+=2), NO
    - for(int i = 0, i <= list.size() - 1; i++), OK
    - for(int i = 0, i <= list.size(); i++), ArrayIndexOutOfBoundsException

Assignment 4

- getting input from the console
  - make it as machine friendly as possible
    - check each line entered for multiple entries with delimiters
    - keep the order consistent with the assignment description
  - make it as human friendly as possible
    - display text that describes the expected format of the input


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
<<interface>>
SortedSet
HashSet
LinkedHashSet
TreeSet
ArrayList
Vector
LinkedList
PriorityQueue
Object
Arrays
Collections
<<interface>>
Map
<<interface>>
SortedMap
Hashtable
LinkedHashMap
HashMap
TreeMap
implements
extends



Unique Values

Constant Time Access

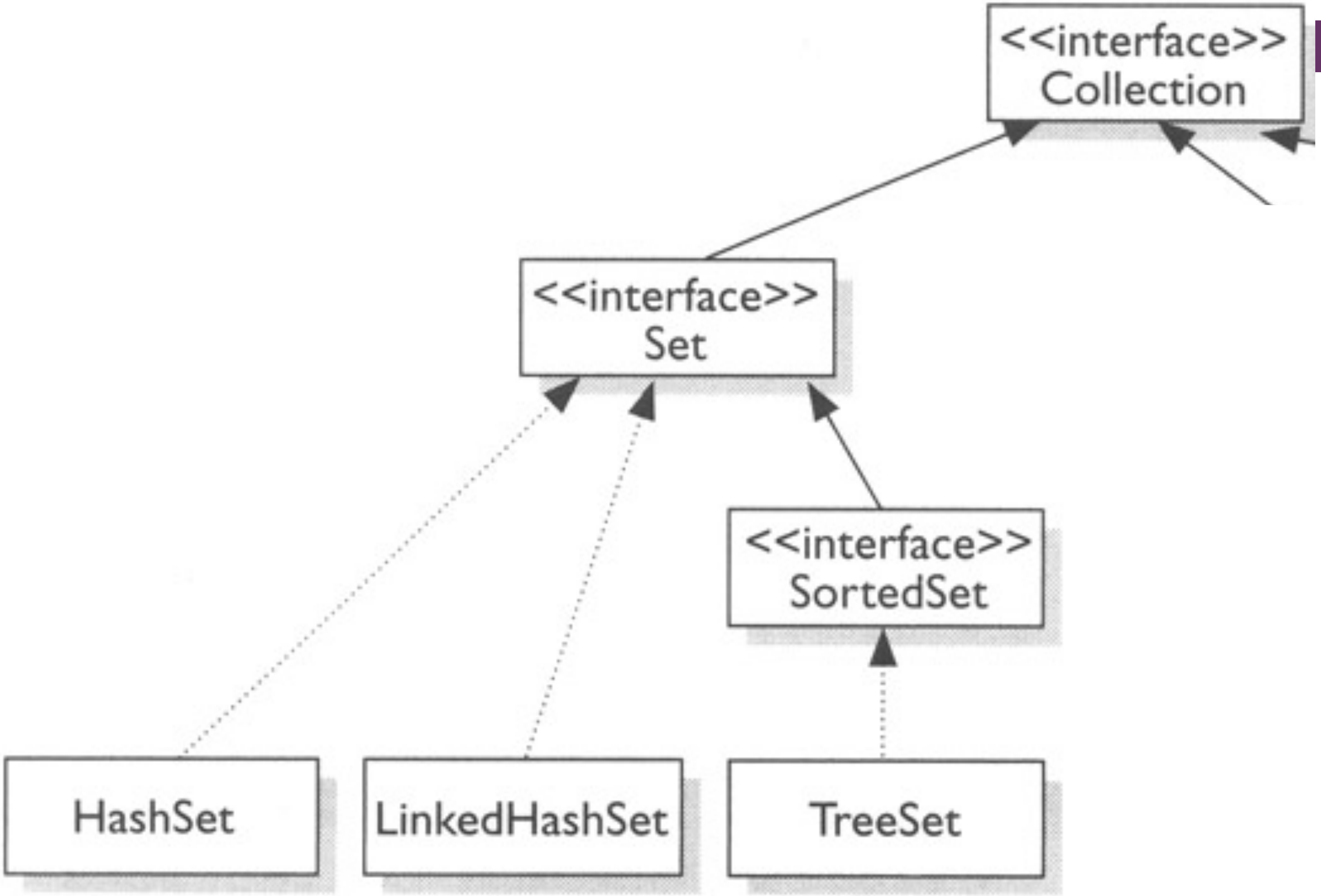
Logarithmic Access

Linear access

Constant Time Access

Sequences

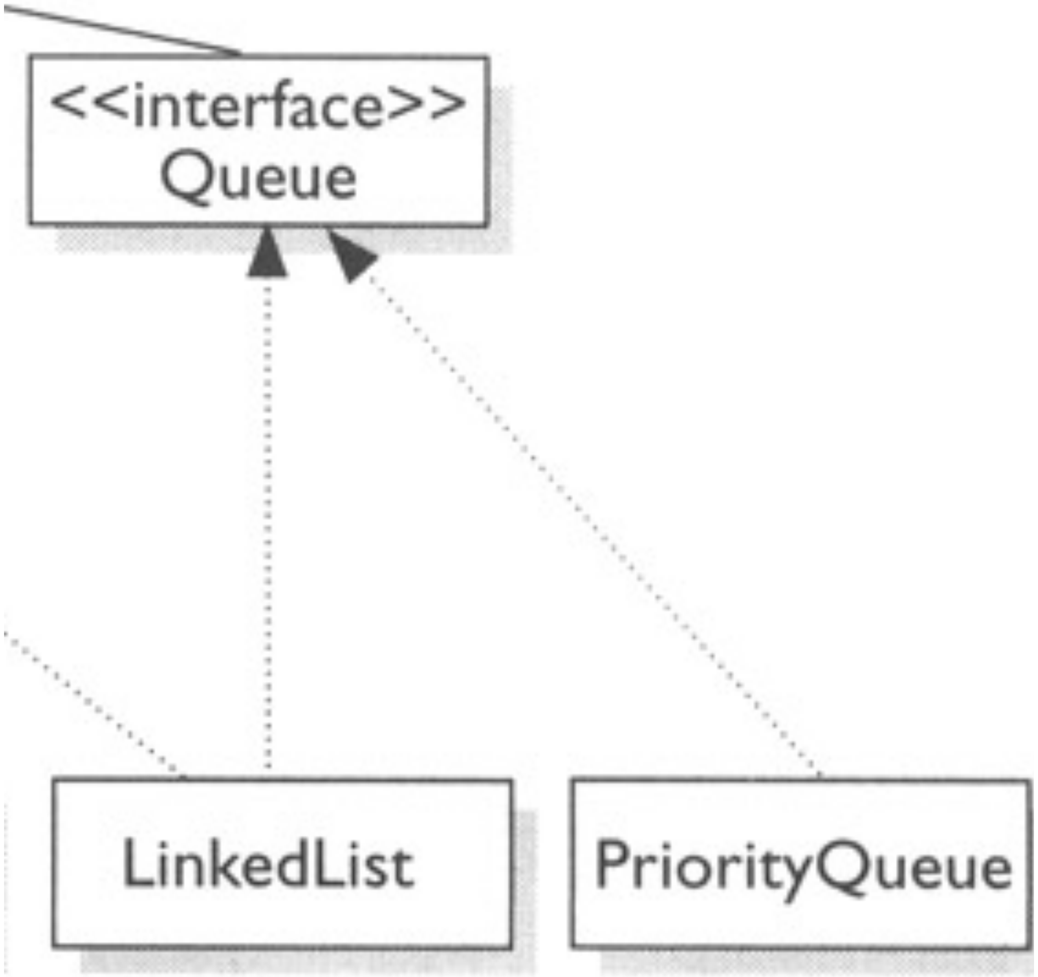
Unique Keys
Key-Value Pairs
Dictionary



Unordered

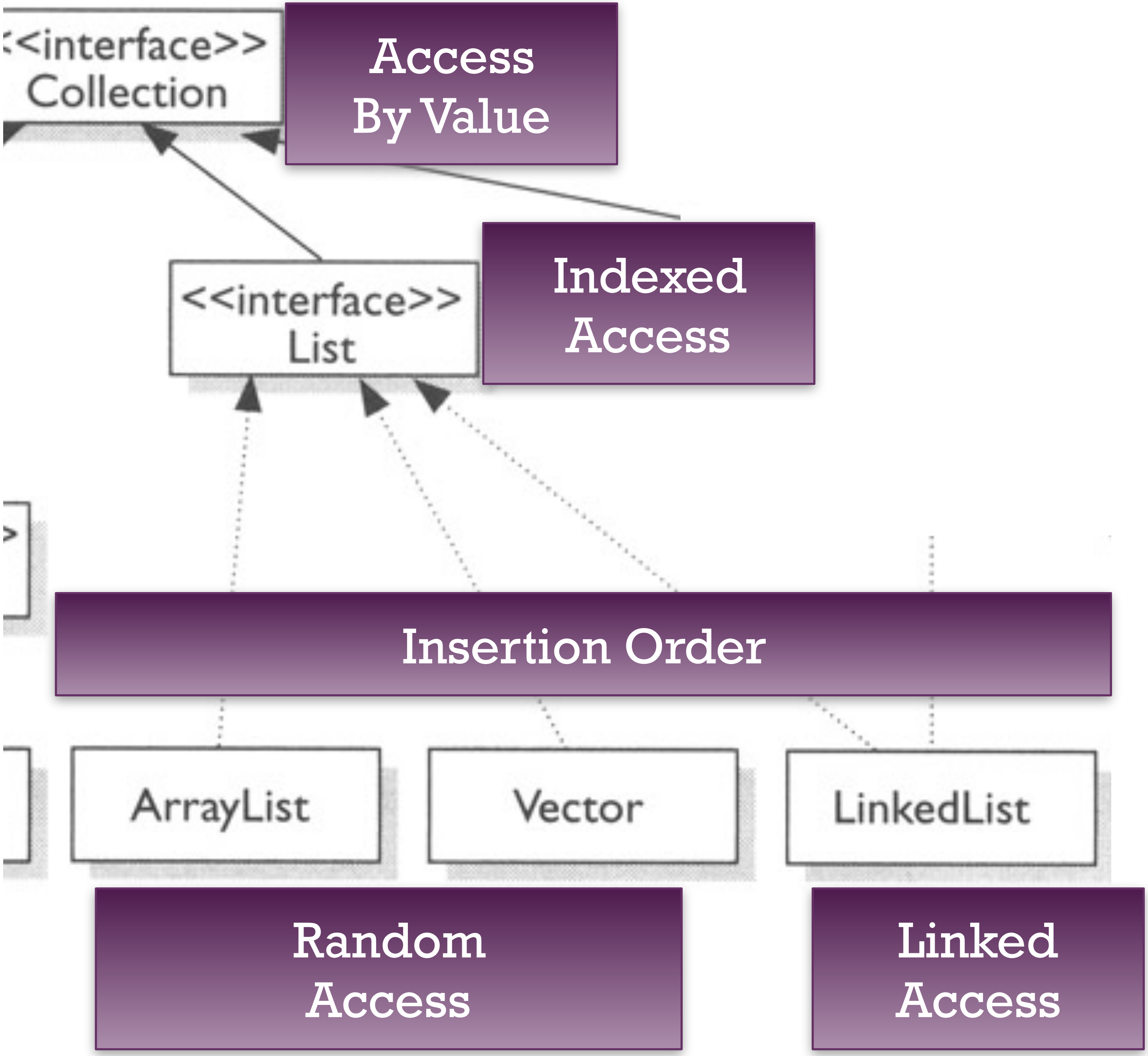
Insertion Order

Natural Order



FIFO Access

Natural Order Access



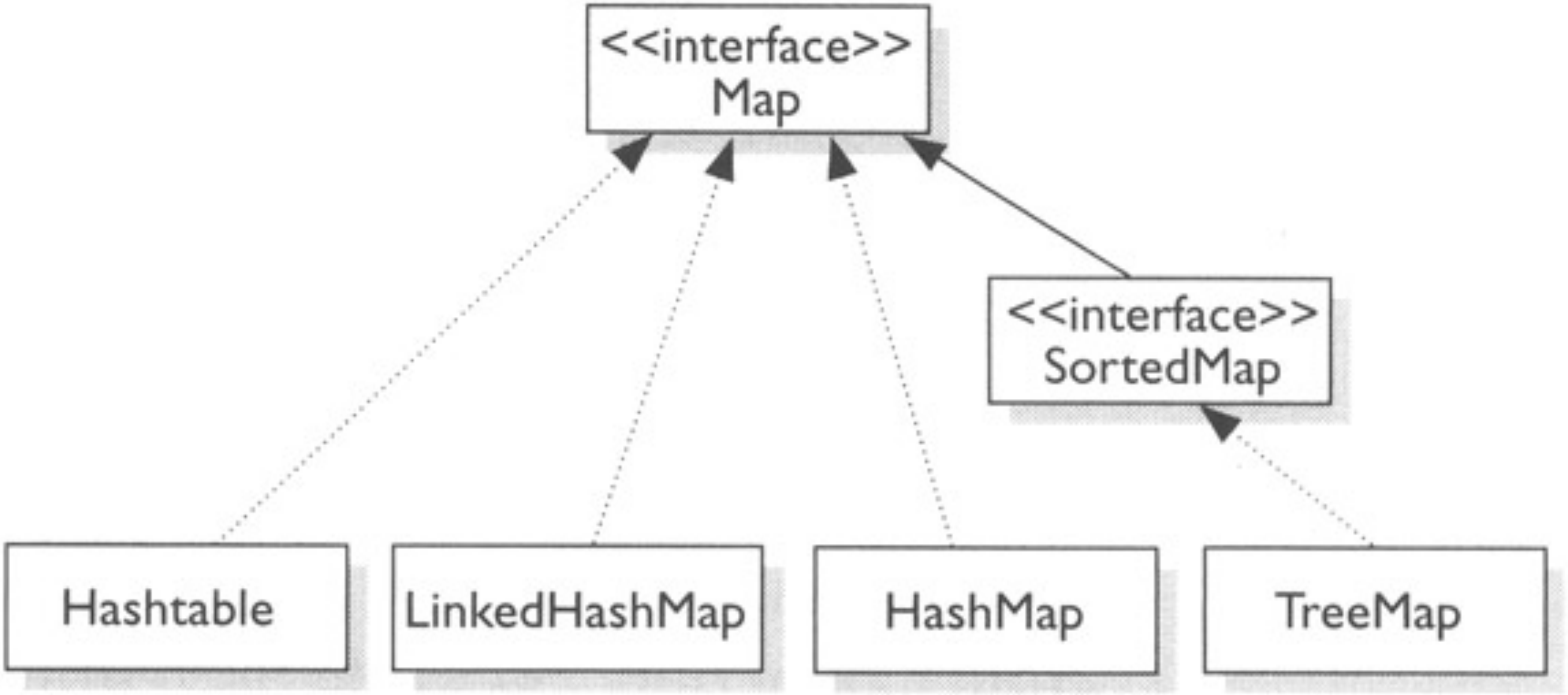
Access By Value

Indexed Access

Insertion Order

Random Access

Linked Access



Unordered
Synchronized

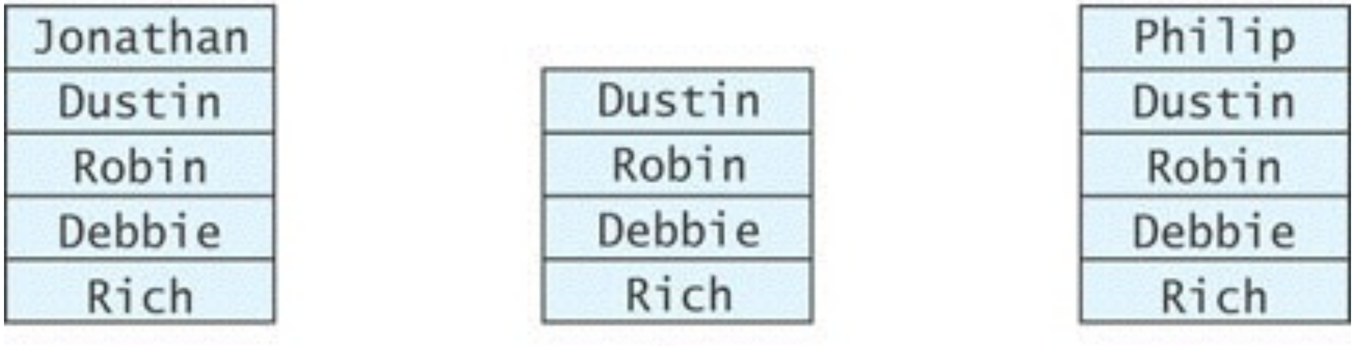
Insertion
Order

Unordered

Natural
Order

Questions?

+ A Stack of Strings



(a) (b) (c)

- “Rich” is the oldest element on the stack and “Jonathan” is the youngest (Figure a)

- `String last = names.peek();` stores a reference to “Jonathan” in `last`

- `String temp = names.pop();` removes “Jonathan” and stores a reference to it in `temp` (Figure b)

- `names.push(“Philip”);` pushes “Philip” onto the stack (Figure c)

+ Finding Palindromes

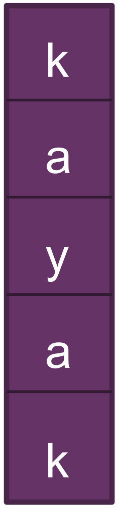
- Palindrome: a string that reads identically in either direction, letter by letter (ignoring case)
  - kayak
  - "I saw I was I"
  - “Able was I ere I saw Elba”
  - "Level madam level"

- Problem: Write a program that reads a string and determines whether it is a palindrome

+ Finding Palindromes (cont.)

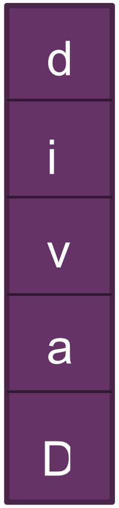
- Solving using a stack:
  - Push each string character, from left to right, onto a stack

"kayak" becomes



k
a
y
a
k

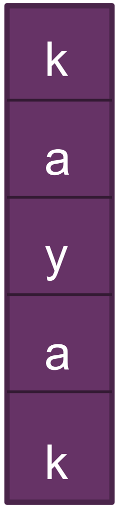
"David" becomes



d
i
v
a
D

+ Finding Palindromes (cont.)

- Solving using a stack:
  - Pop each character off the stack, appending each to the `StringBuilder result`



k

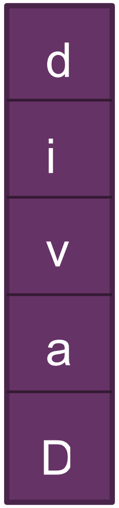
a

y

a

k

becomes "kayak"



d

i

v

a

D

becomes "divaD"

+ Balanced Parentheses

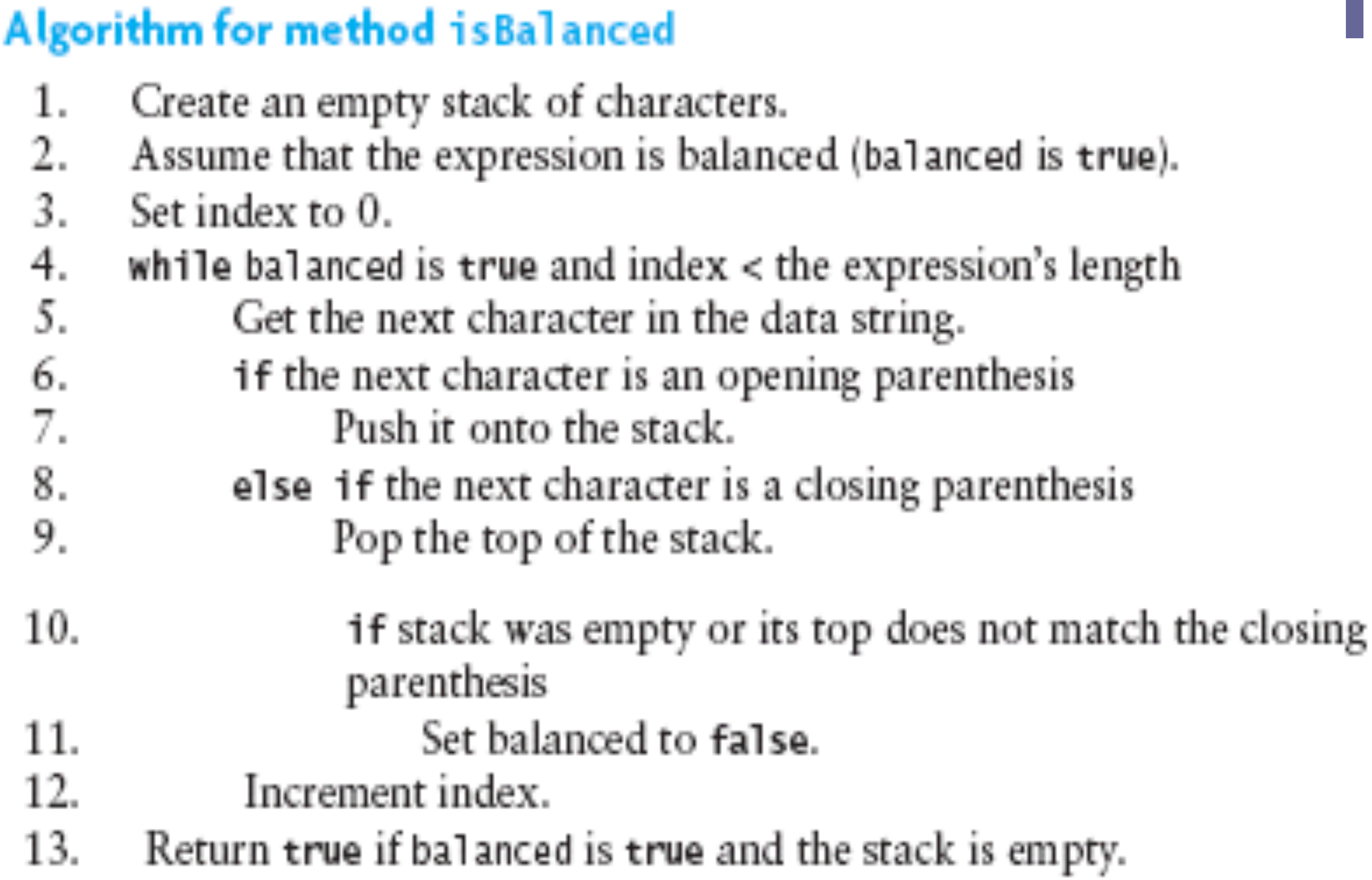
- When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parentheses

```
( a + b * ( c / ( d – e ) ) ) + ( d / e )
```

- The problem is further complicated if braces or brackets are used in conjunction with parentheses

- The solution is to use stacks!

+ Balanced Parentheses (cont.)

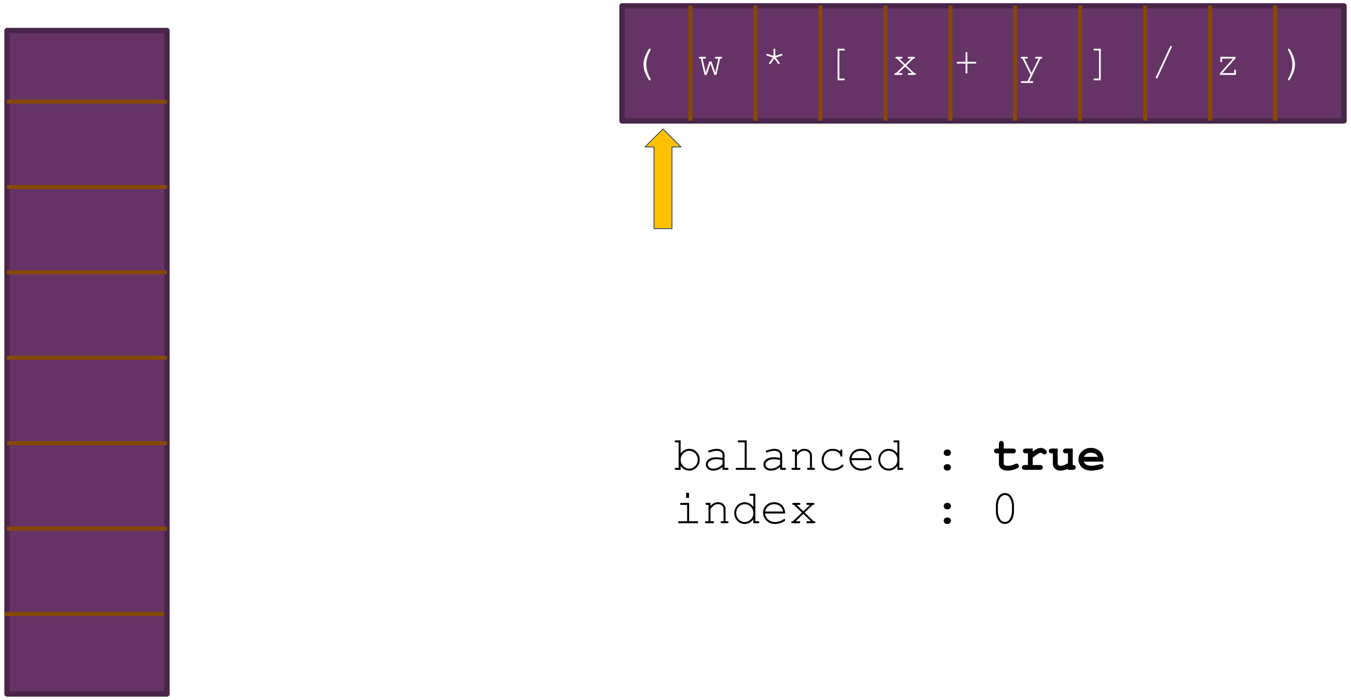


Algorithm for method isBalanced

```
Create an empty stack of characters.
Assume that the expression is balanced (balanced is true).
Set index to 0.
while balanced is true and index < the expression's length
    Get the next character in the data string.
    if the next character is an opening parenthesis
        Push it onto the stack.
    else if the next character is a closing parenthesis
        Pop the top of the stack.
        if stack was empty or its top does not match the closing
        parenthesis
            Set balanced to false.
    Increment index.
Return true if balanced is true and the stack is empty.
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`

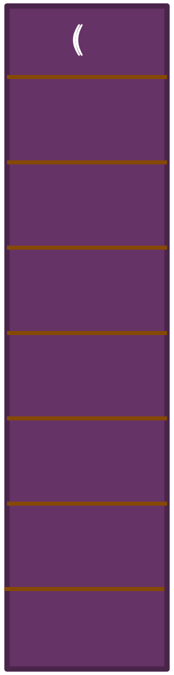


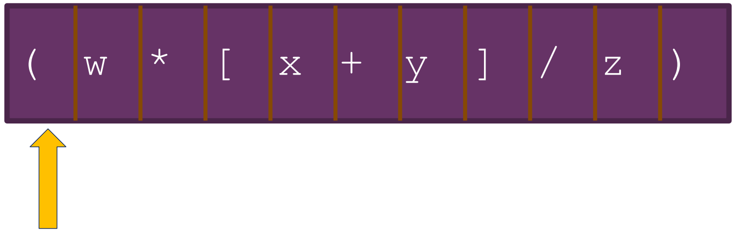
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 0
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`

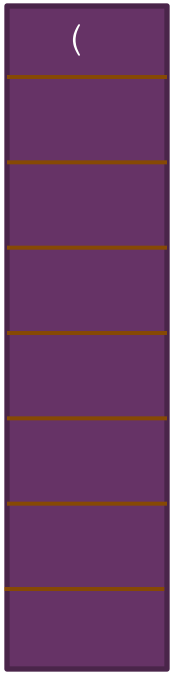


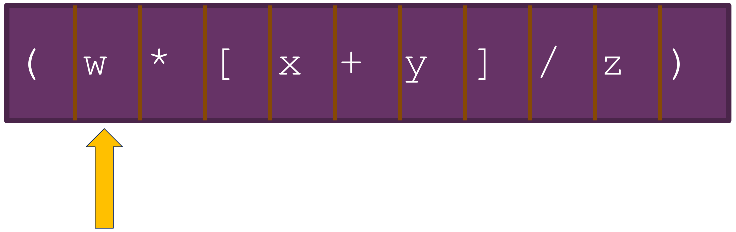


```
balanced : true
index    : 0
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



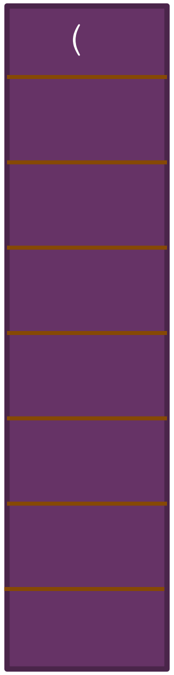


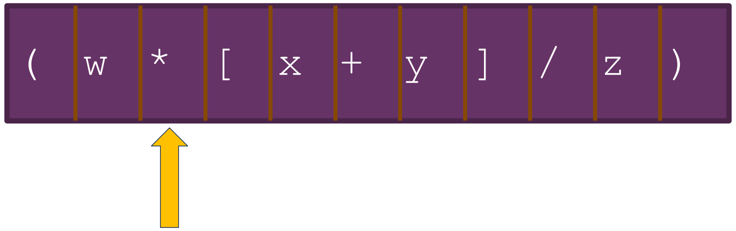
(	w	*	[	x	+	y	]	/	z	)

```
balanced : **true**
index    : 1
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`

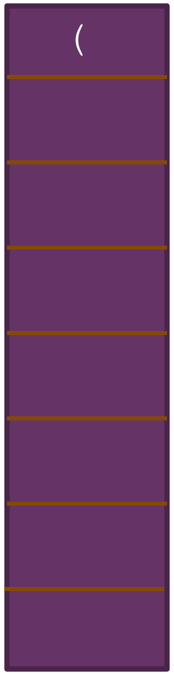




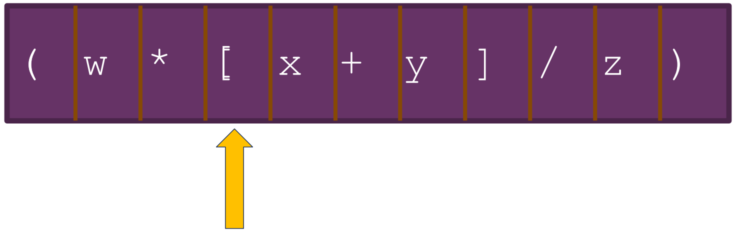
```
balanced : true
index    : 2
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



(

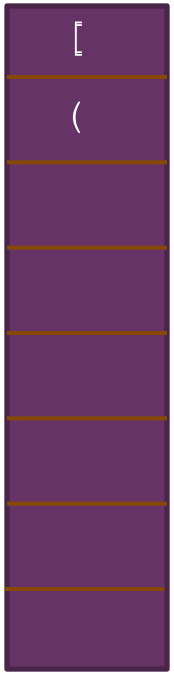


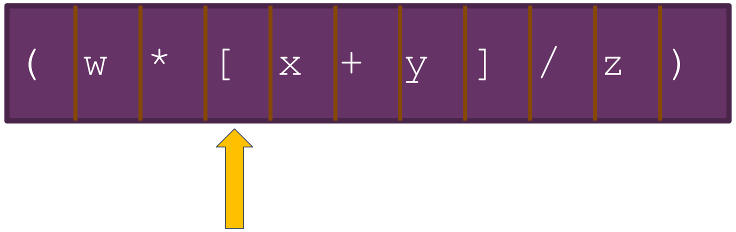
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 3
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`

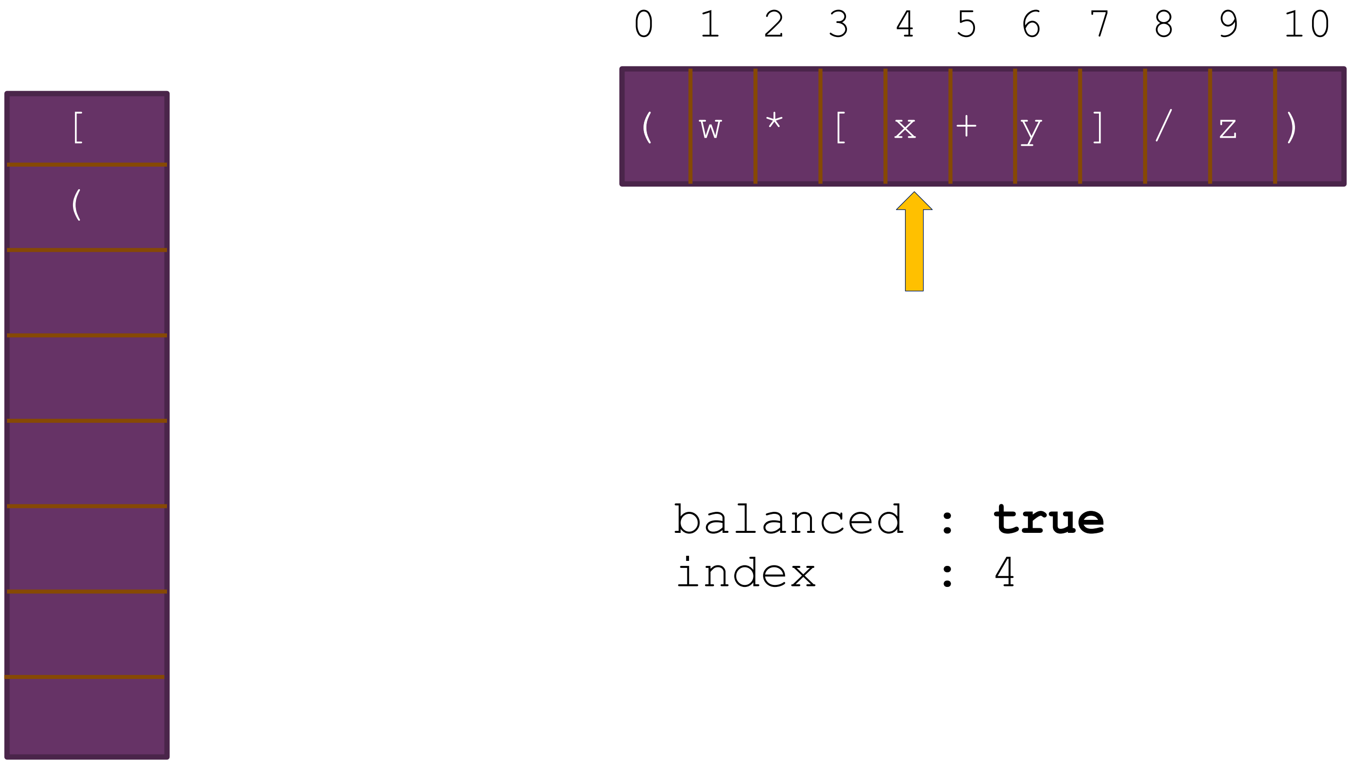




```
balanced : true
index    : 3
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



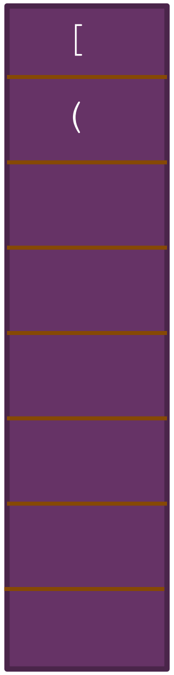
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

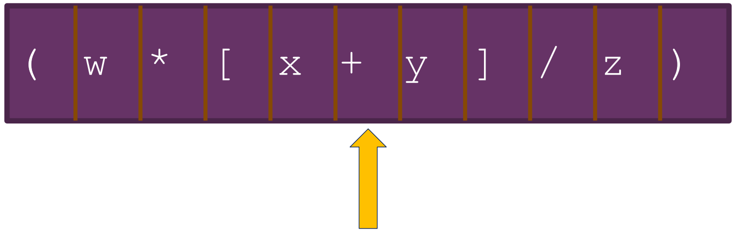
[
(

```
balanced : true
index    : 4
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`

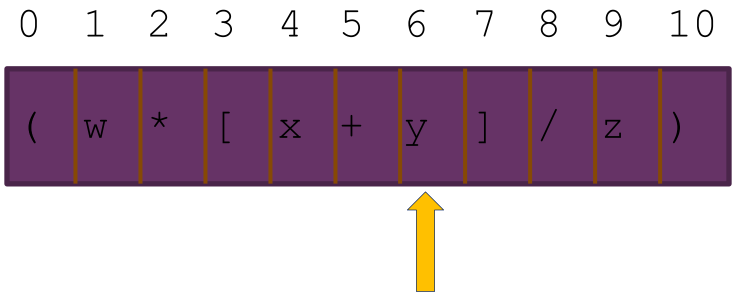




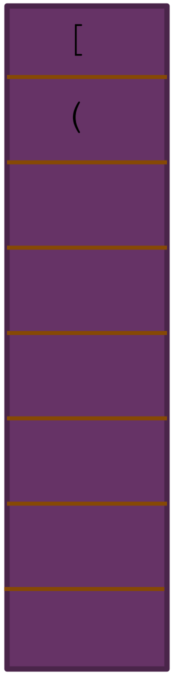
```
balanced : true
index    : 5
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

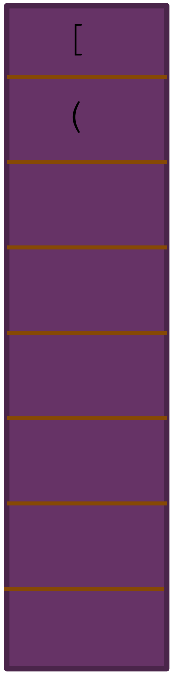


[
(

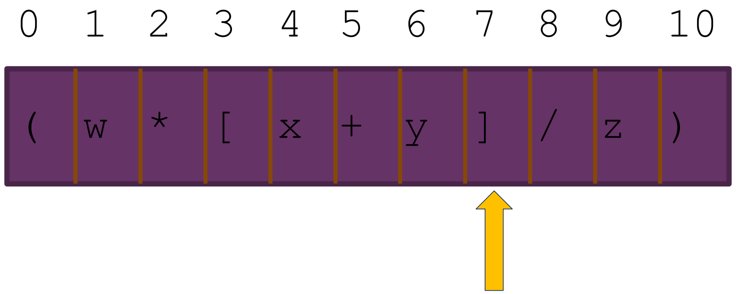
```
balanced : true
index    : 6
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



```
[
(
```

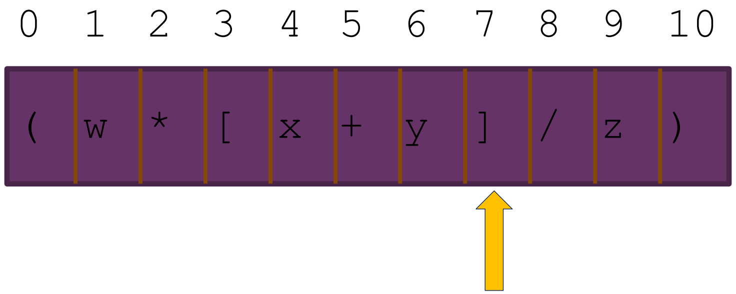


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

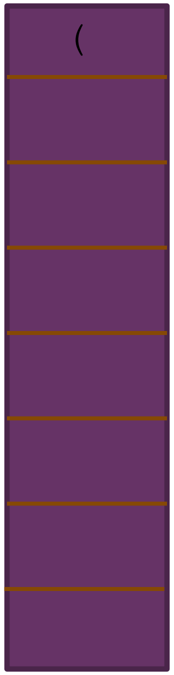
```
balanced : true
index    : 7
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)



(

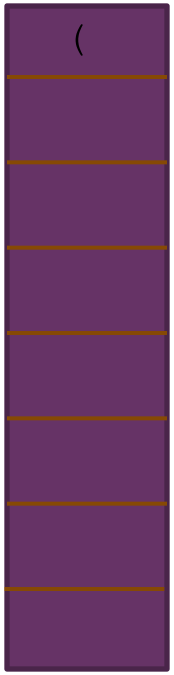
[

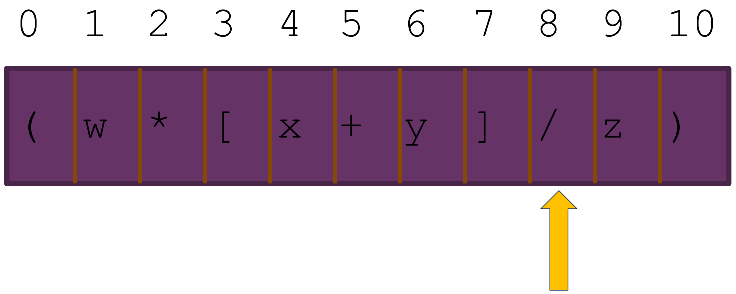
Matches!
Balanced **still** true

```
balanced : true
index    : 7
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



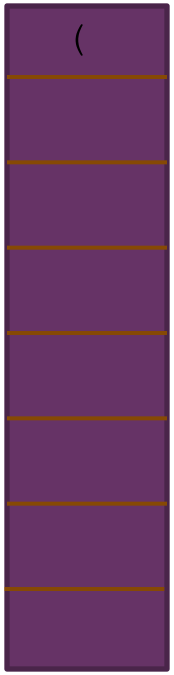


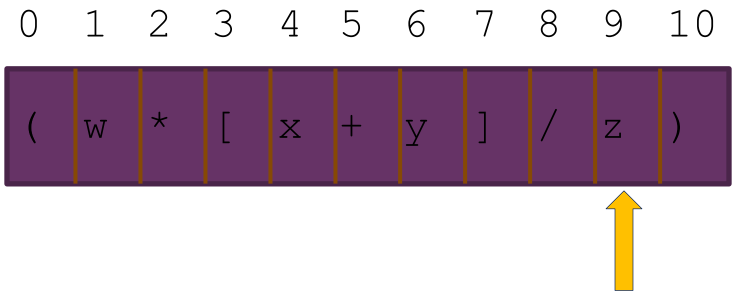
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 8
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



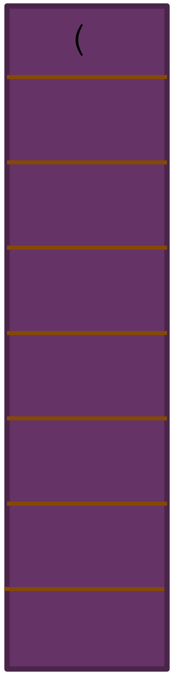


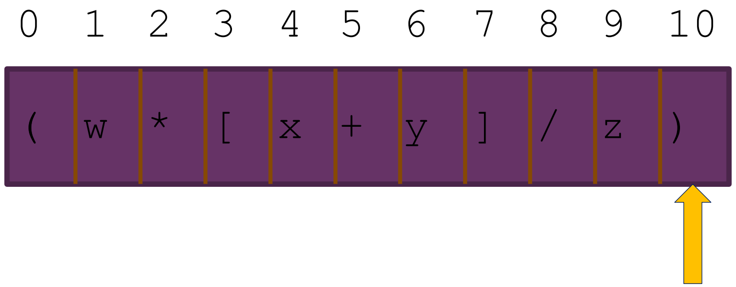
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 9
```

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



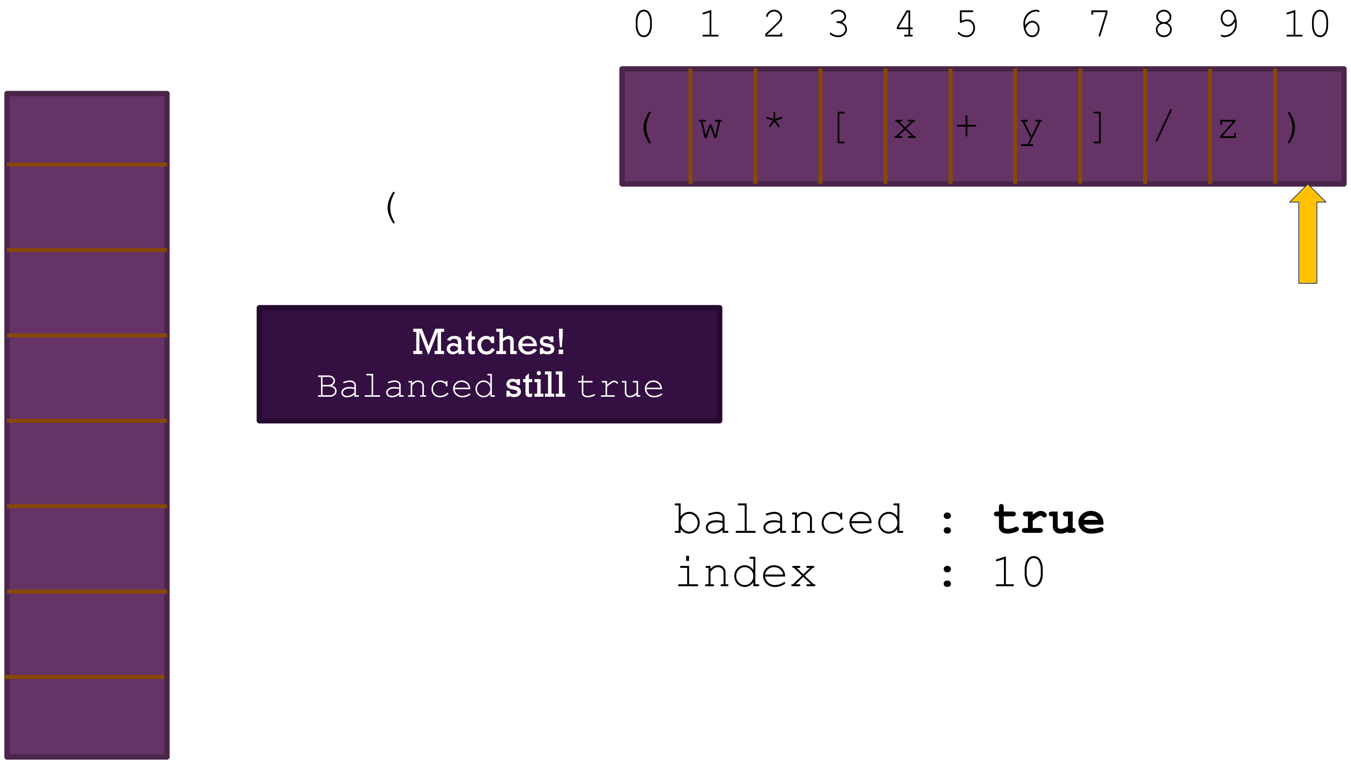


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 10
```

+ Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

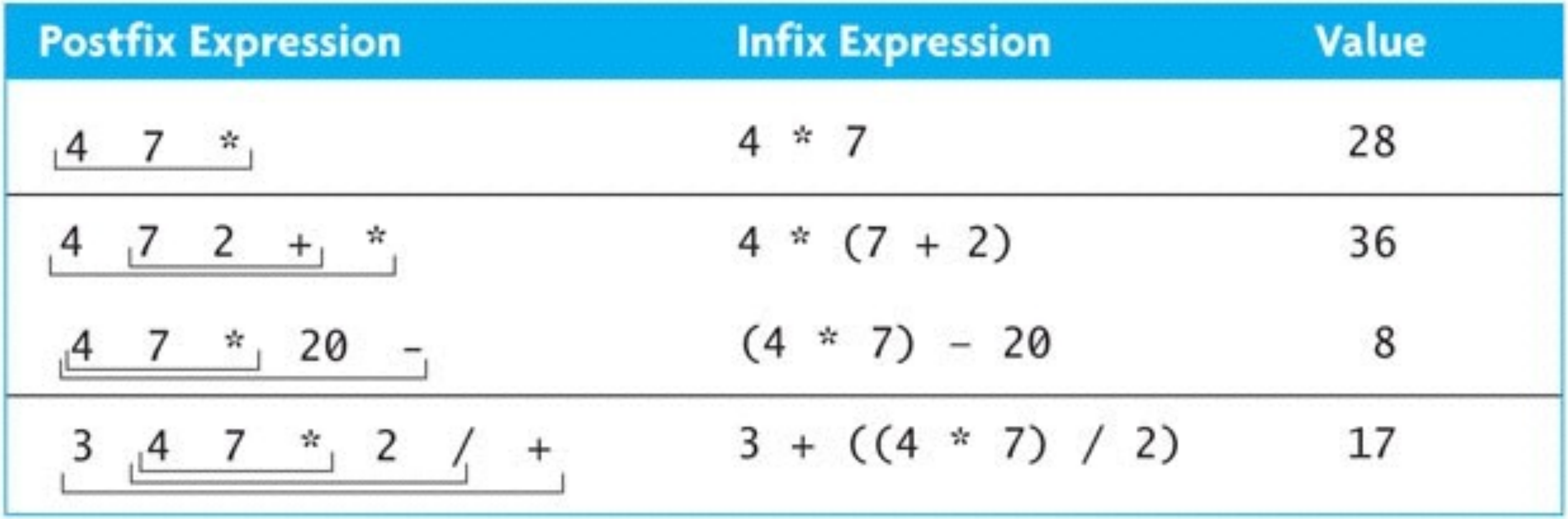
(

Matches!
`Balanced` **still** `true`

```
balanced : true
index    : 10
```

Additional Stack Applications

- Postfix and infix notation
  - Expressions normally are written in infix form, but
  - it easier to evaluate an expression in postfix form since there is no need to group sub-expressions in parentheses or worry about operator precedence



Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 -	(4 * 7) - 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

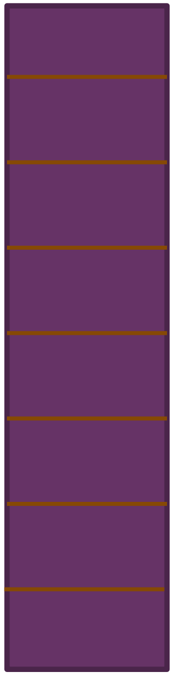
+ Evaluating Postfix Expressions (cont.)

4	7	*	20	-

```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

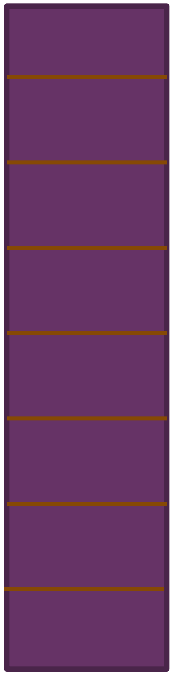
+ Evaluating Postfix Expressions (cont.)

4	7	*	20	-



```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

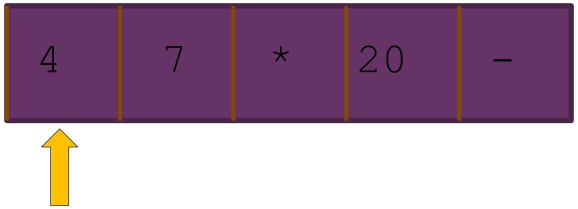
Evaluating Postfix Expressions (cont.)

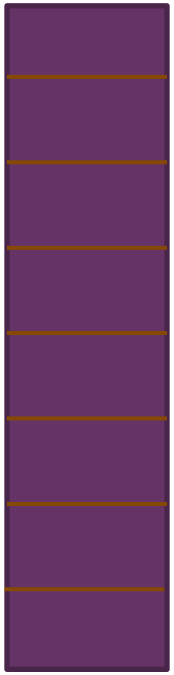


4	7	*	20	-

```
   create an empty stack of integers
➡  while there are more tokens
     get the next token
     if the first character of the token is a digit
        push the token on the stack
     else if the token is an operator
        pop the right operand off the stack
        pop the left operand off the stack
        evaluate the operation
        push the result onto the stack
   pop the stack and return the result
```

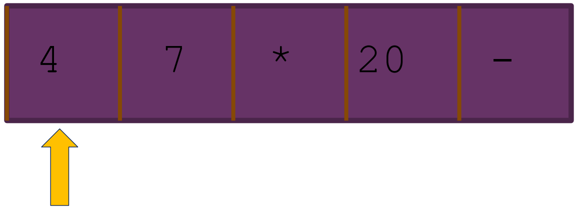
Evaluating Postfix Expressions (cont.)



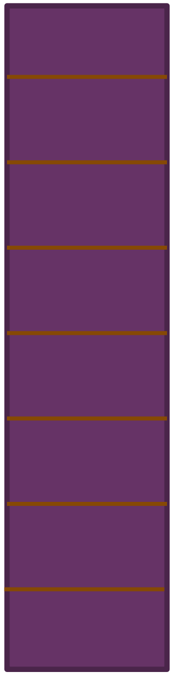


```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)

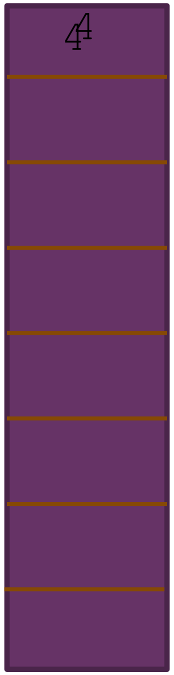


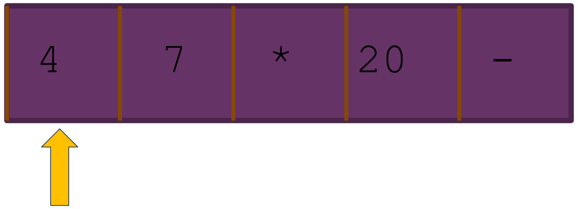
4	7	*	20	-



```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

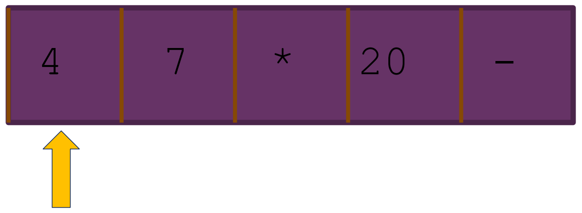
+ Evaluating Postfix Expressions (cont.)

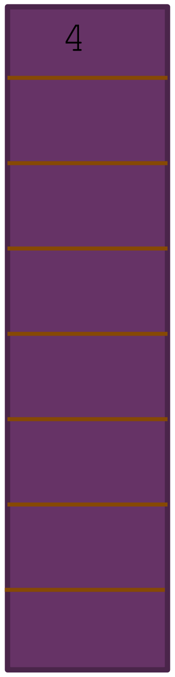




```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

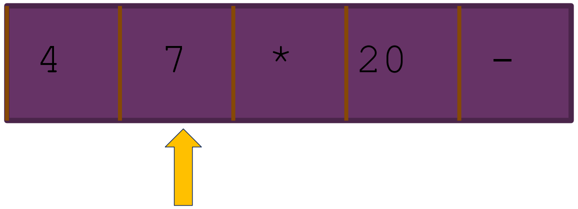
+ Evaluating Postfix Expressions (cont.)

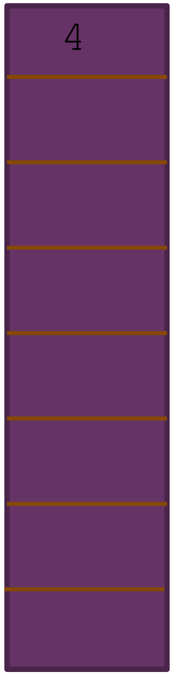




```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

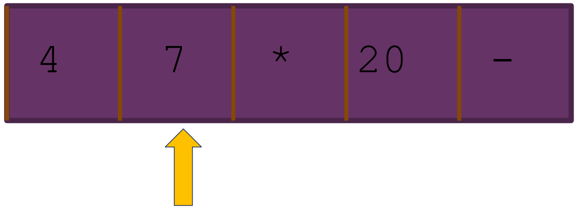
+ Evaluating Postfix Expressions (cont.)

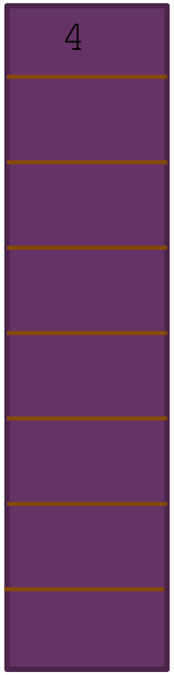




```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

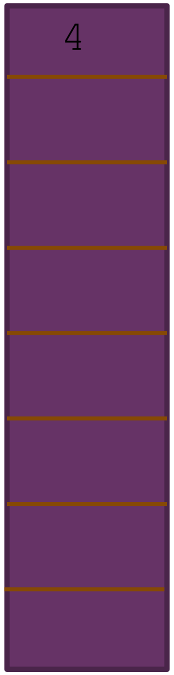
+ Evaluating Postfix Expressions (cont.)



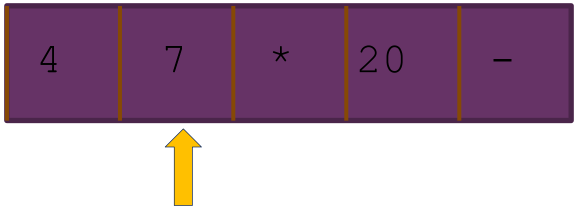


```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

Evaluating Postfix Expressions (cont.)



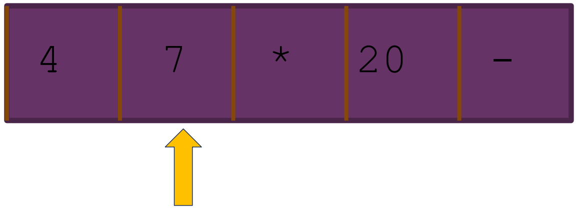
4

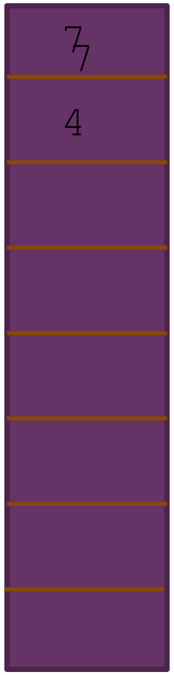


4 7 * 20 -

```
create an empty stack of integers
while there are more tokens
   get the next token
   if the first character of the token is a digit
      push the token on the stack
   else if the token is an operator
      pop the right operand off the stack
      pop the left operand off the stack
      evaluate the operation
      push the result onto the stack
pop the stack and return the result
```

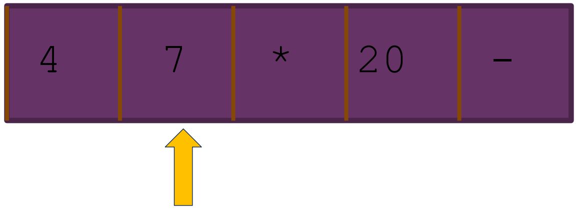
+ Evaluating Postfix Expressions (cont.)

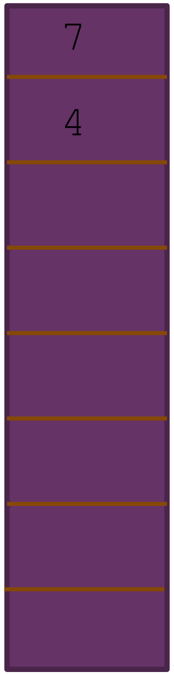




```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

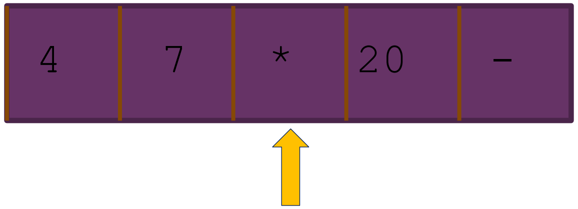
+ Evaluating Postfix Expressions (cont.)

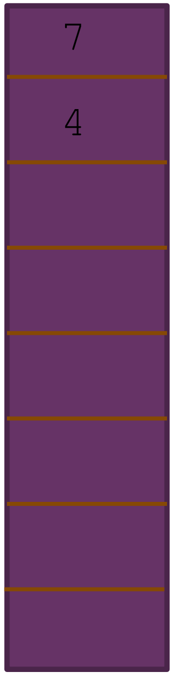




```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

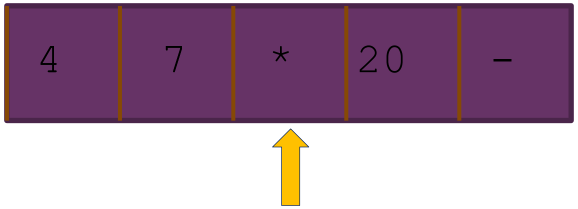
+ Evaluating Postfix Expressions (cont.)



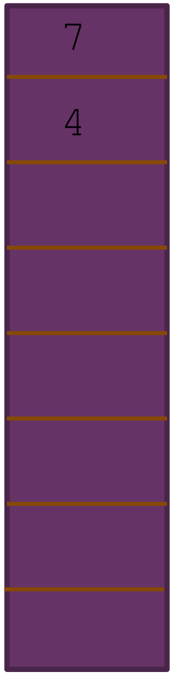


```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)



4 7 * 20 -

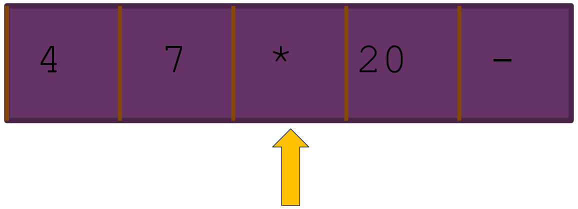


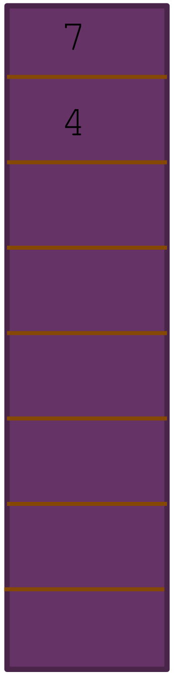
7

4

```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)

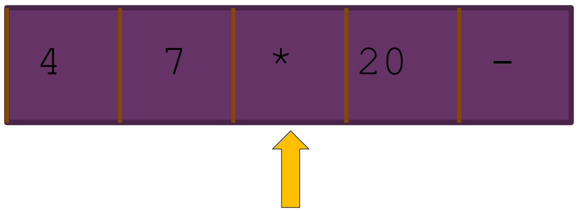


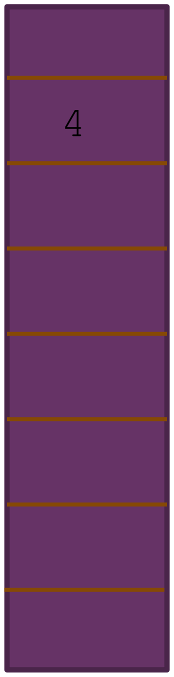


```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)

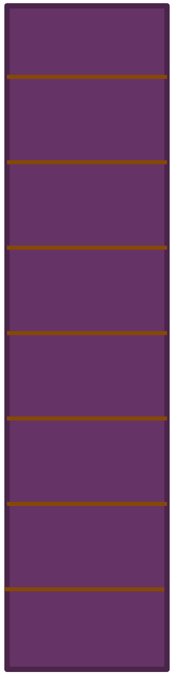
7



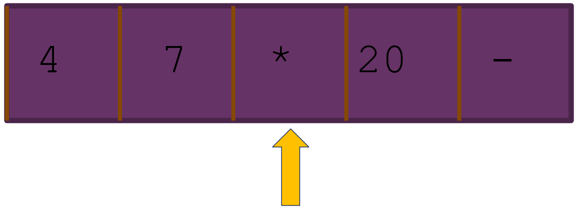


```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)



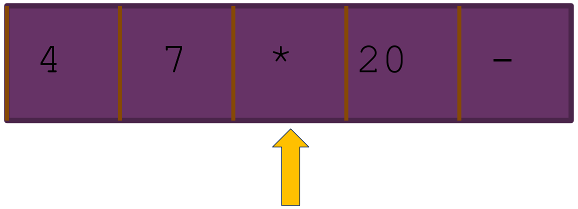
4 7



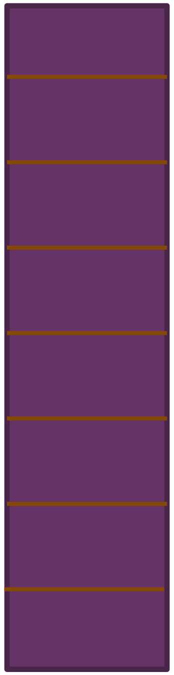
```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)

4 * 7

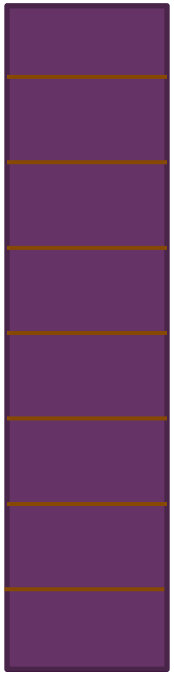


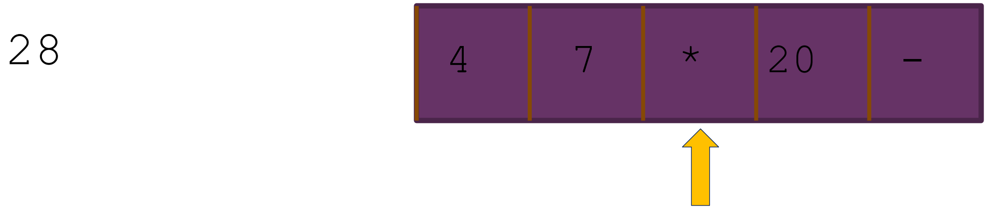
4	7	*	20	-



```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

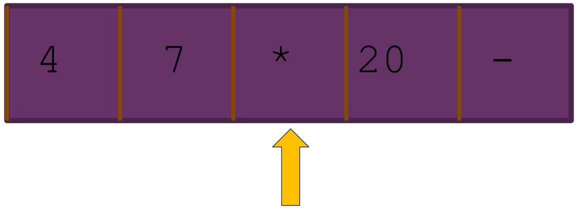
+ Evaluating Postfix Expressions (cont.)

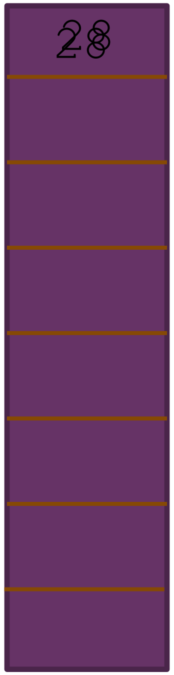




```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

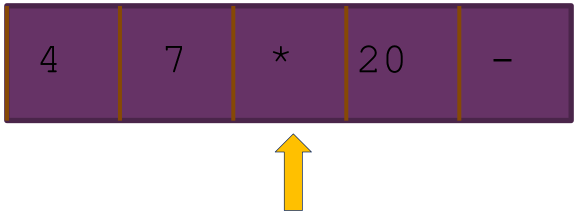
+ Evaluating Postfix Expressions (cont.)

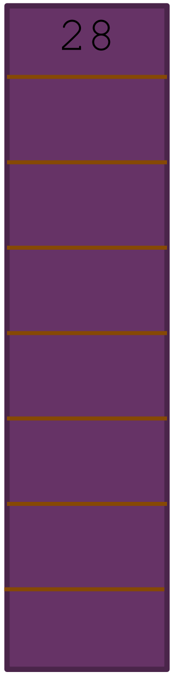




```
create an empty stack of integers
while there are more tokens
   get the next token
   if the first character of the token is a digit
      push the token on the stack
   else if the token is an operator
      pop the right operand off the stack
      pop the left operand off the stack
      evaluate the operation
      push the result onto the stack
pop the stack and return the result
```

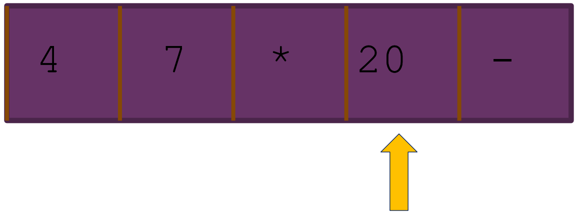
+ Evaluating Postfix Expressions (cont.)

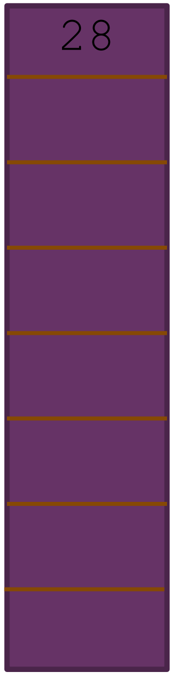




```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

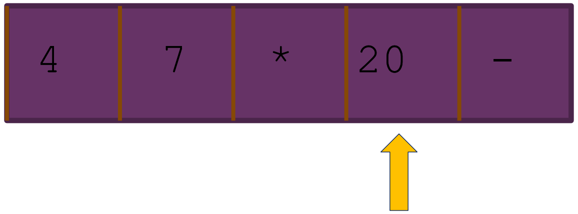
+ Evaluating Postfix Expressions (cont.)

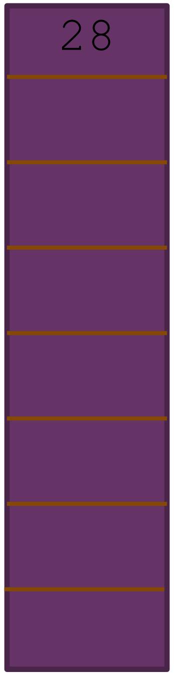




```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

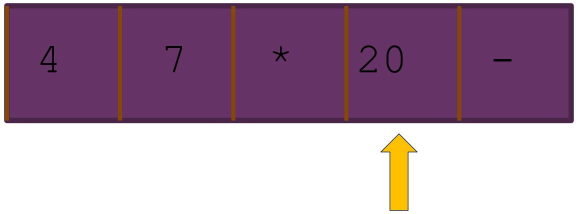
+ Evaluating Postfix Expressions (cont.)

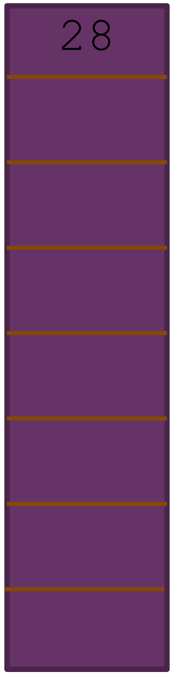




```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

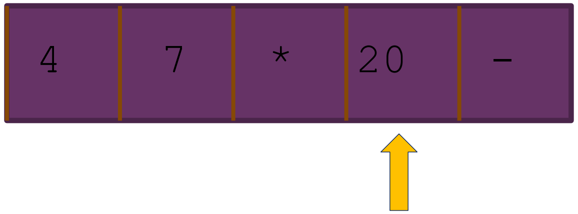
+ Evaluating Postfix Expressions (cont.)

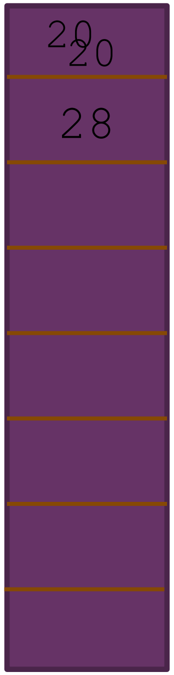




```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

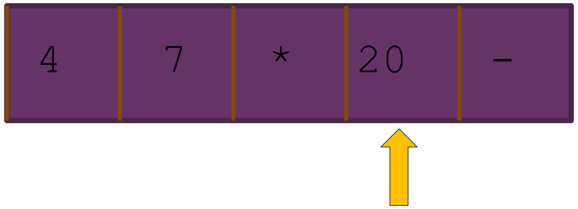
+ Evaluating Postfix Expressions (cont.)



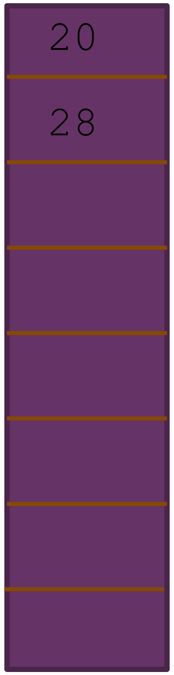


```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Evaluating Postfix Expressions (cont.)



4 7 * 20 -

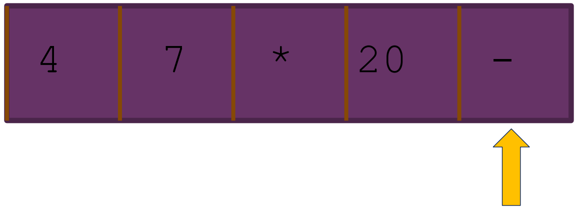


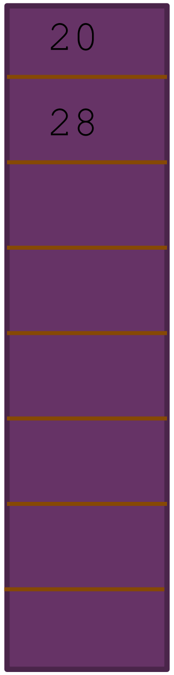
20

28

```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

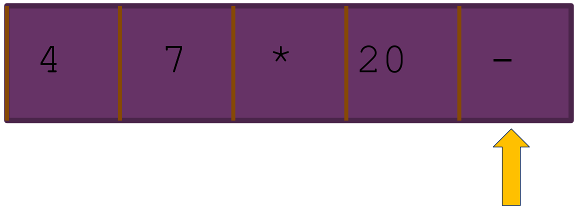
+ Evaluating Postfix Expressions (cont.)

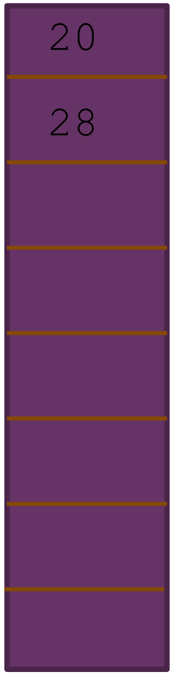




```
create an empty stack of integers
while there are more tokens
   get the next token
   if the first character of the token is a digit
      push the token on the stack
   else if the token is an operator
      pop the right operand off the stack
      pop the left operand off the stack
      evaluate the operation
      push the result onto the stack
pop the stack and return the result
```

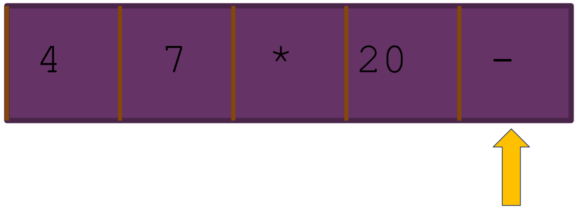
+ Evaluating Postfix Expressions (cont.)

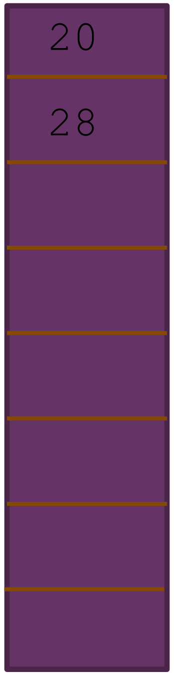




```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

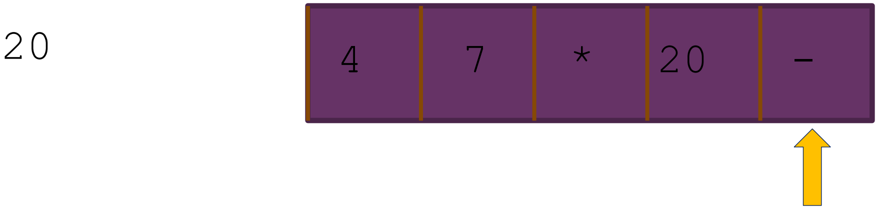
+ Evaluating Postfix Expressions (cont.)

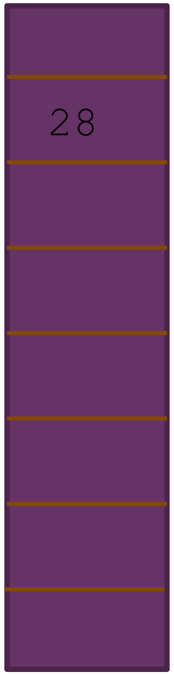

4
7
*
20
-


20
28

```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

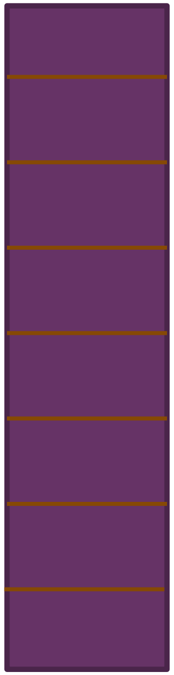
+ Evaluating Postfix Expressions (cont.)



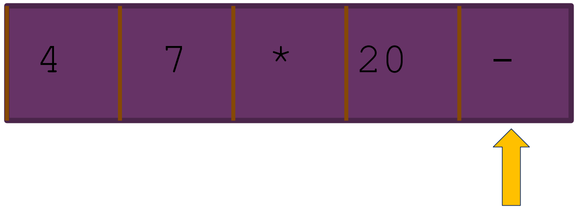


```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)



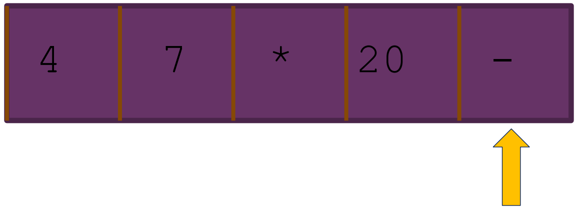
28 20

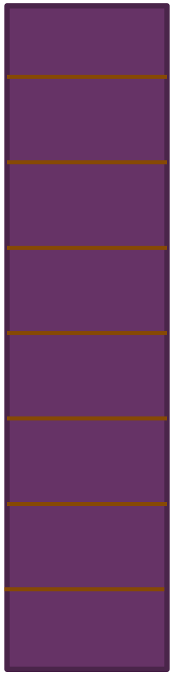


```
create an empty stack of integers
while there are more tokens
   get the next token
   if the first character of the token is a digit
      push the token on the stack
   else if the token is an operator
      pop the right operand off the stack
      pop the left operand off the stack
      evaluate the operation
      push the result onto the stack
pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)

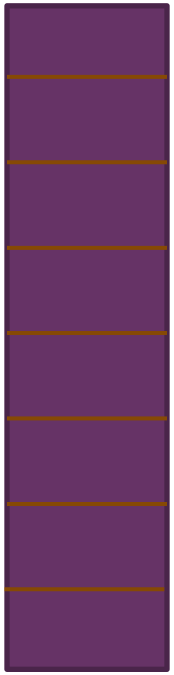
28 - 20



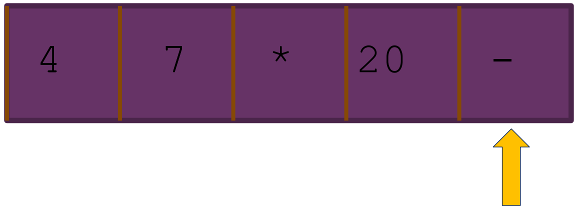


```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)



8


4

7

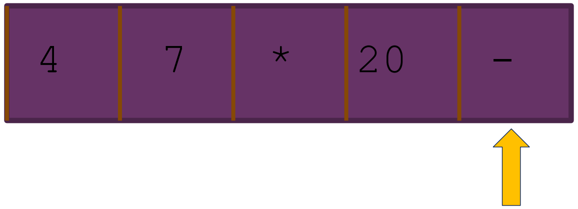
*

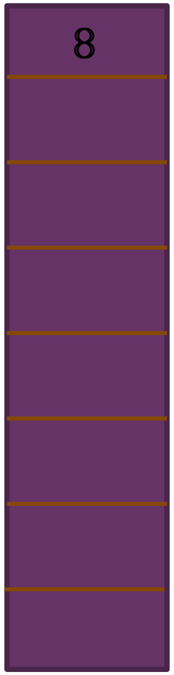
20

-

```
create an empty stack of integers
while there are more tokens
   get the next token
   if the first character of the token is a digit
      push the token on the stack
   else if the token is an operator
      pop the right operand off the stack
      pop the left operand off the stack
      evaluate the operation
      push the result onto the stack
pop the stack and return the result
```

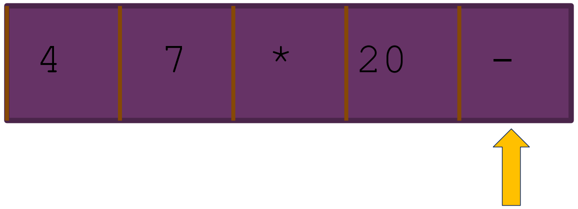
+ Evaluating Postfix Expressions (cont.)

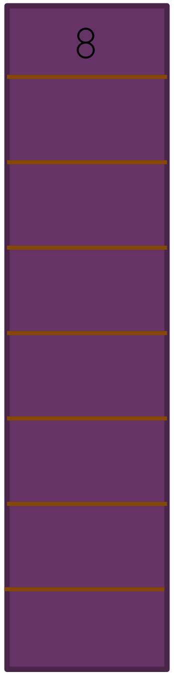




```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

+ Evaluating Postfix Expressions (cont.)

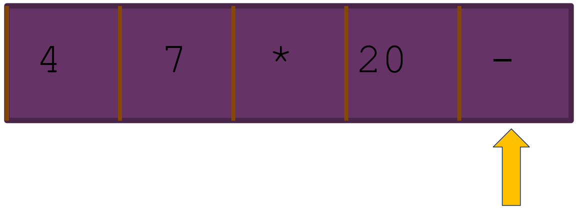


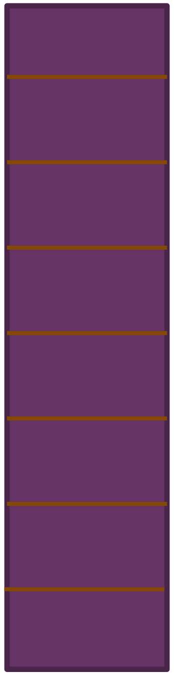


```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

Evaluating Postfix Expressions (cont.)

8





```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

Questions?

+ Java's Stack Implementation

- Extending a `Vector` (as is done by Java) is a poor choice for stack implementation, since all `Vector` methods are accessible

- This is true for extending any Collection based class because the Collection and all sub-interfaces have extra access methods that are inappropriate for a Stack implementation.

- Instead, if you want to use an already available Java class, then you can make an *adapter class* that uses method delegation on the adapted class to implement the Stack methods.

+Implementing a Stack with a List Component

- As an alternative to a stack as an extension of Vector, we can write a class, ListStack, that has a List component (in the example below, theData)

- We can use either the ArrayList, Vector, or the LinkedList classes, as all implement the List interface.

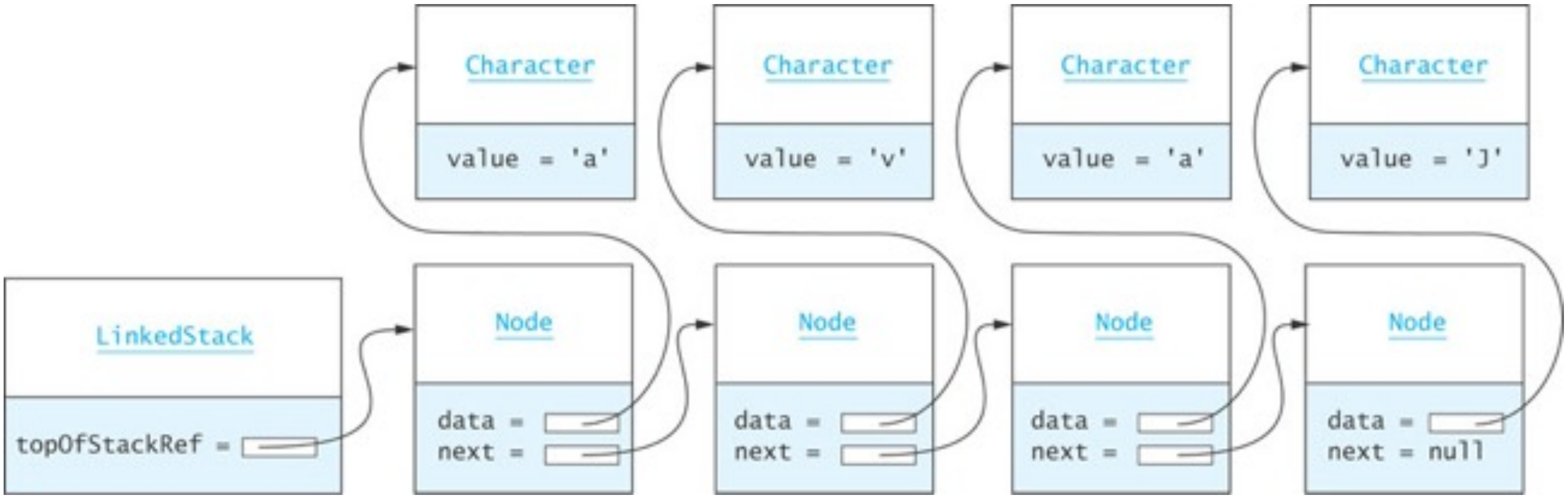
```
public class ListStack<E> {
  List<E> theData;
  public ListStack() {
      theData = new LinkedList<E>();
  }
  …
}
```

- A class which adapts methods of another class by giving different names to essentially the same methods (push instead of add) is called an *adapter class*

- Writing methods in this way is called *method delegation*

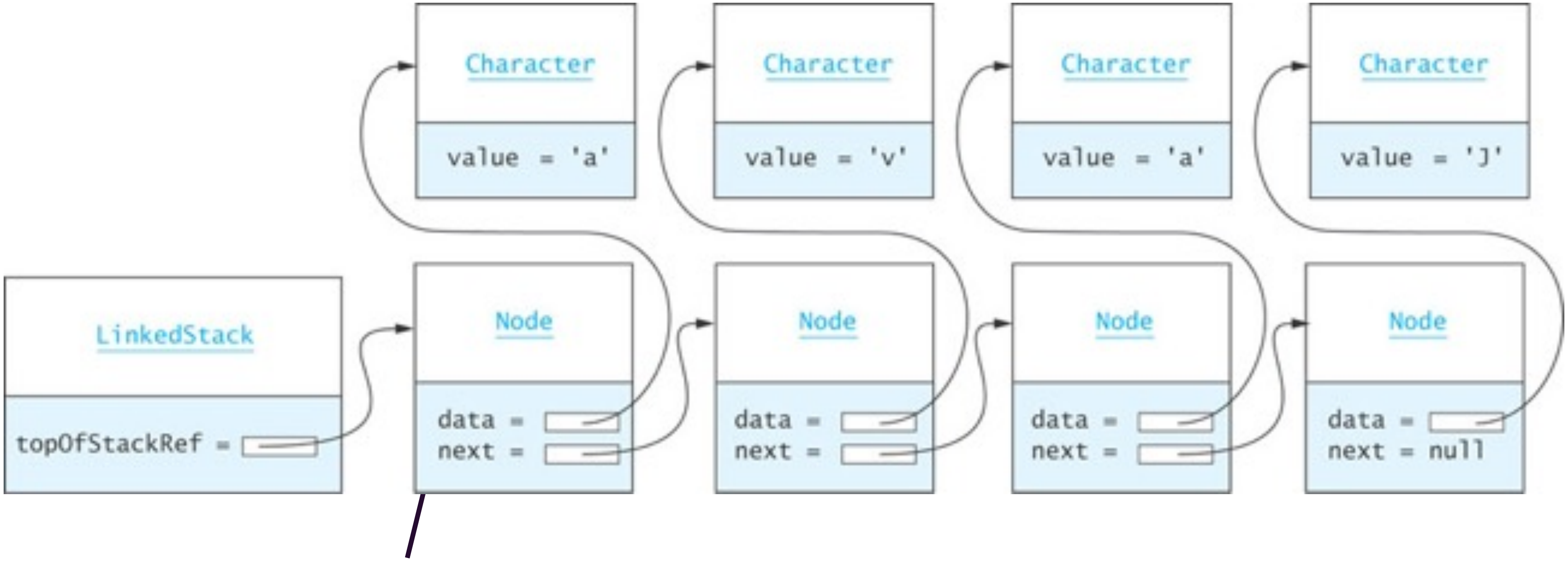
+ Implementing a Stack as a Linked Data Structure

- We can also implement a stack using a linked list of nodes



+ Implementing a Stack as a Linked Data Structure

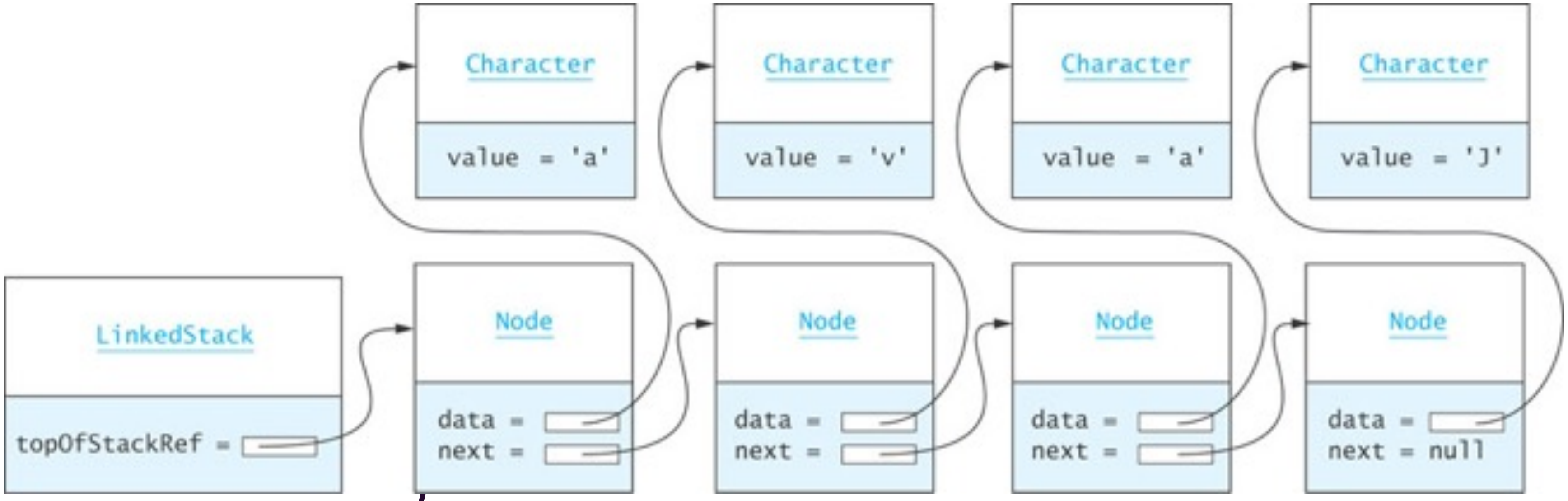
- We can also implement a stack using a linked list of nodes



It is easiest to insert and delete from the head of a list

+ Implementing a Stack as a Linked Data Structure

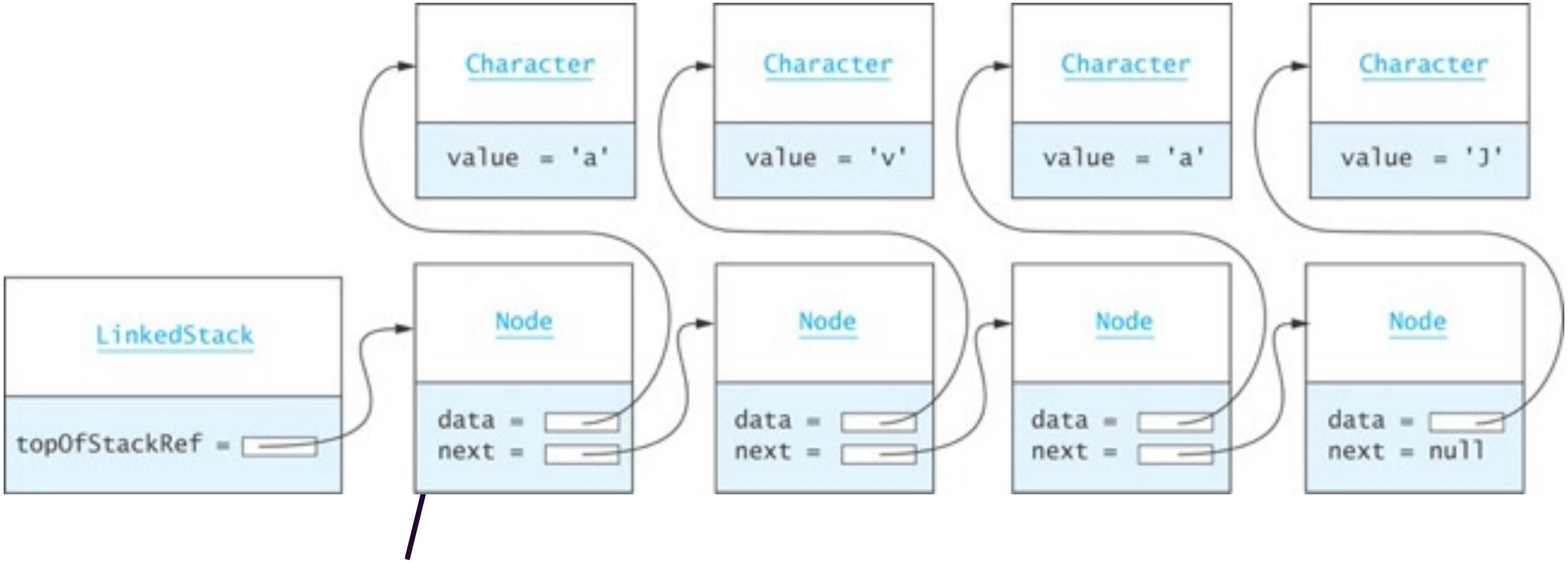
- We can also implement a stack using a linked list of nodes



push inserts a node at the head and pop deletes the node at the head

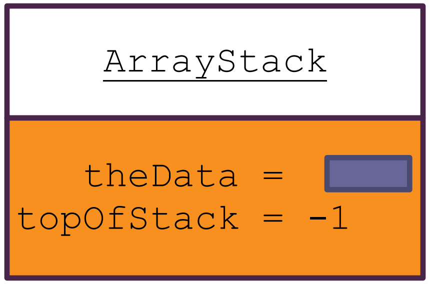
+ Implementing a Stack as a Linked Data Structure

- We can also implement a stack using a linked list of nodes



when the list is empty, pop returns null

Implementing a Stack Using an Array

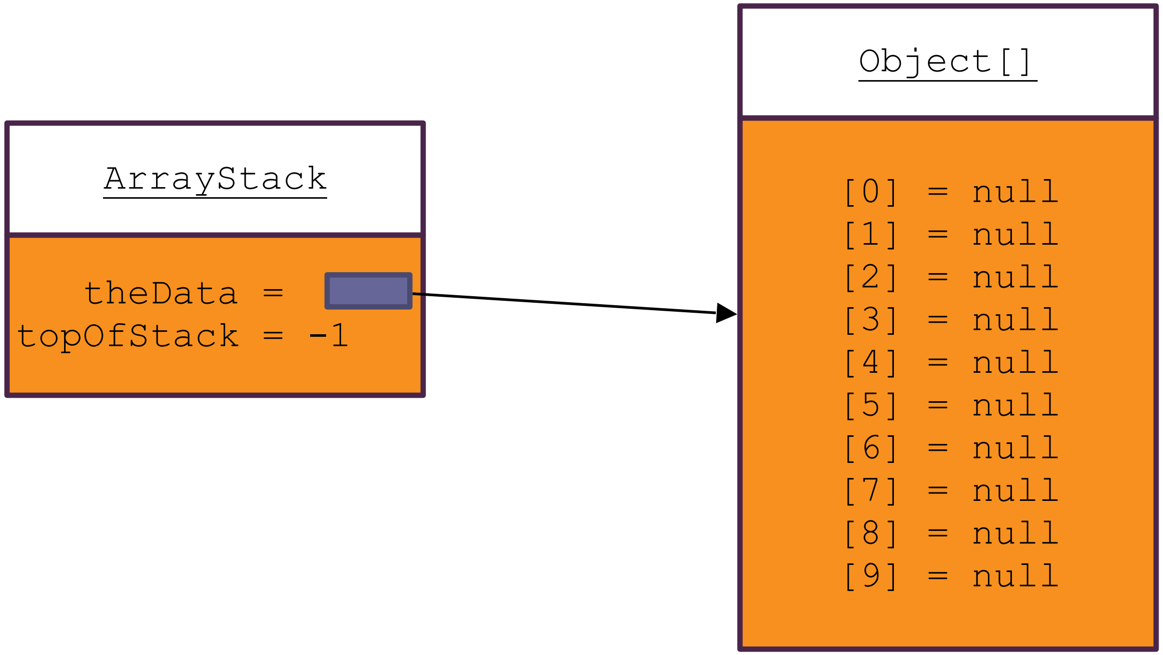


ArrayStack

theData =

topOfStack = -1

Implementing a Stack Using an Array



ArrayStack

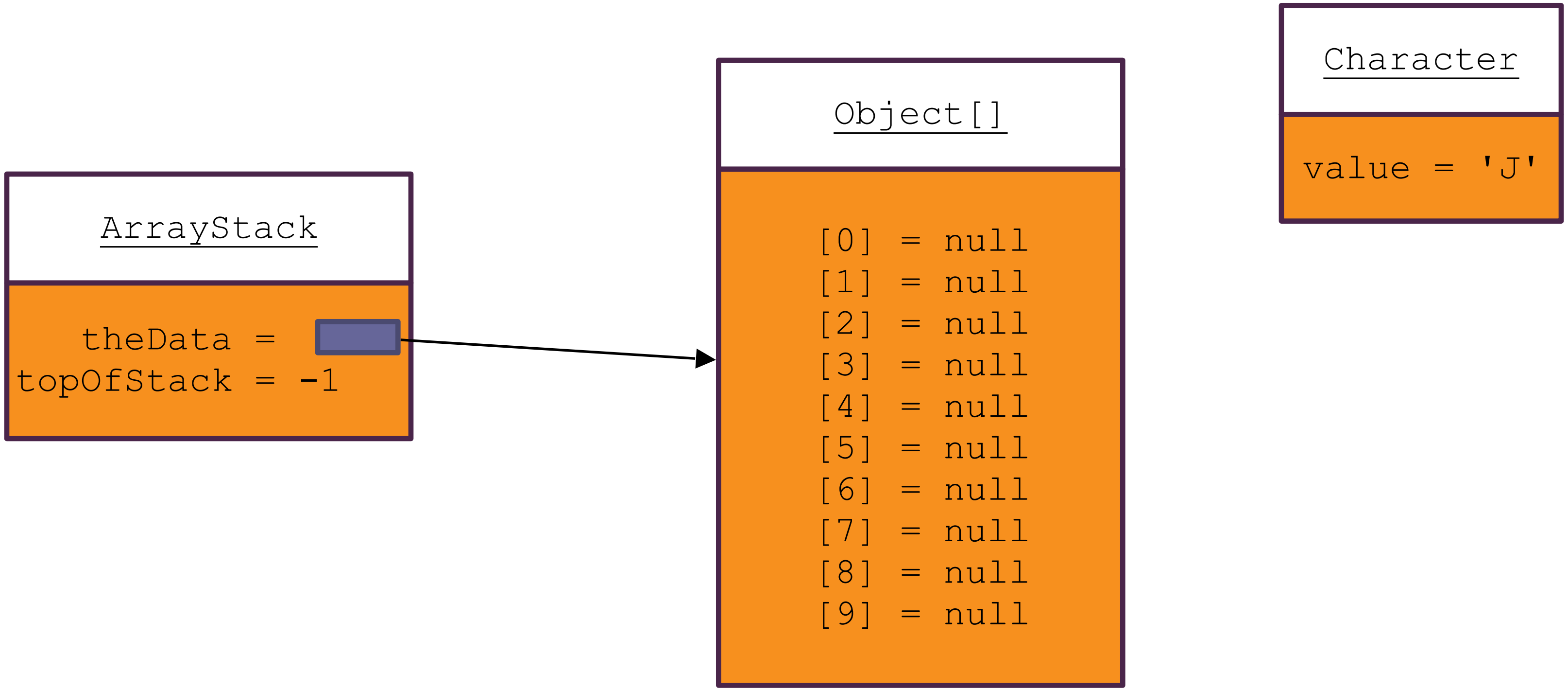
theData =

topOfStack = -1

Object[]

[0] = null
[1] = null
[2] = null
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

Implementing a Stack Using an Array



ArrayStack

theData =
topOfStack = -1

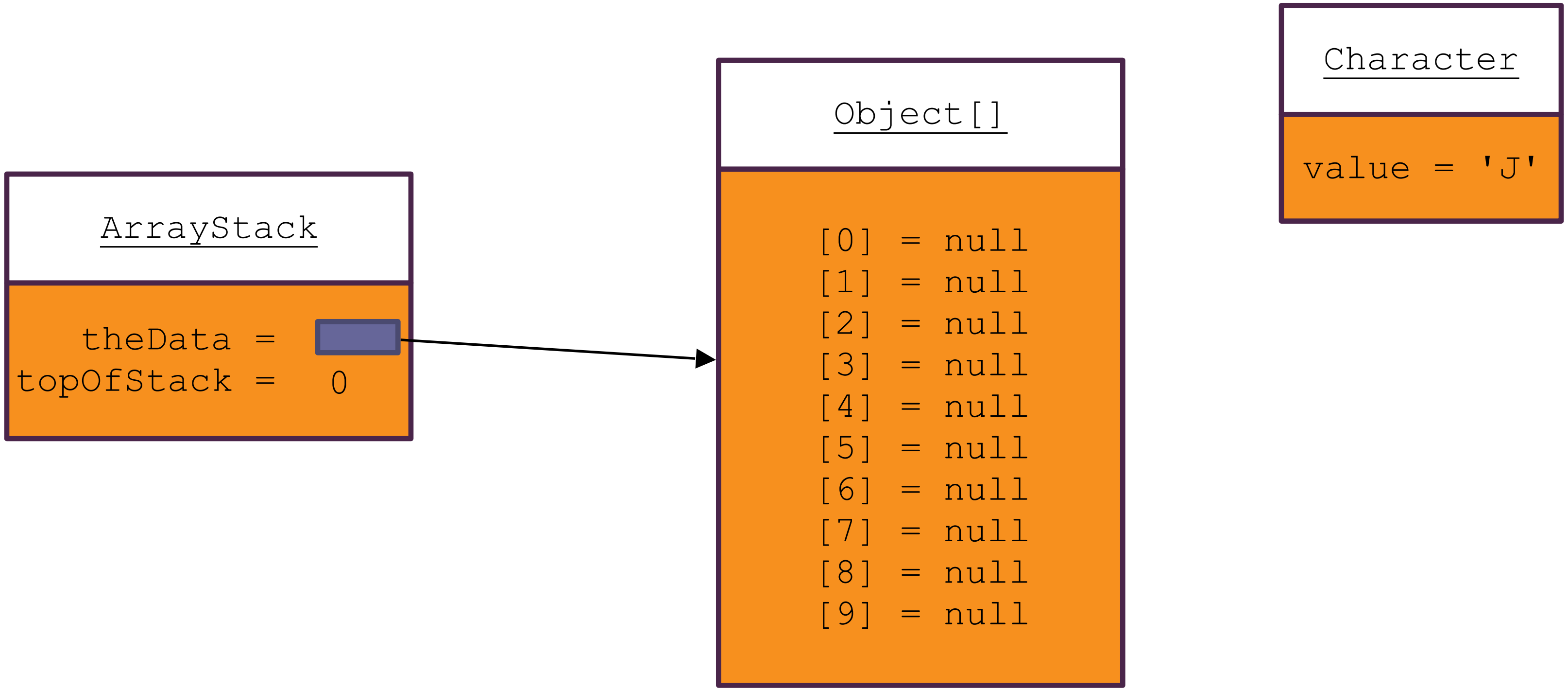
Object[]

[0] = null
[1] = null
[2] = null
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

Character

value = 'J'

Implementing a Stack Using an Array



ArrayStack

theData =
topOfStack = 0

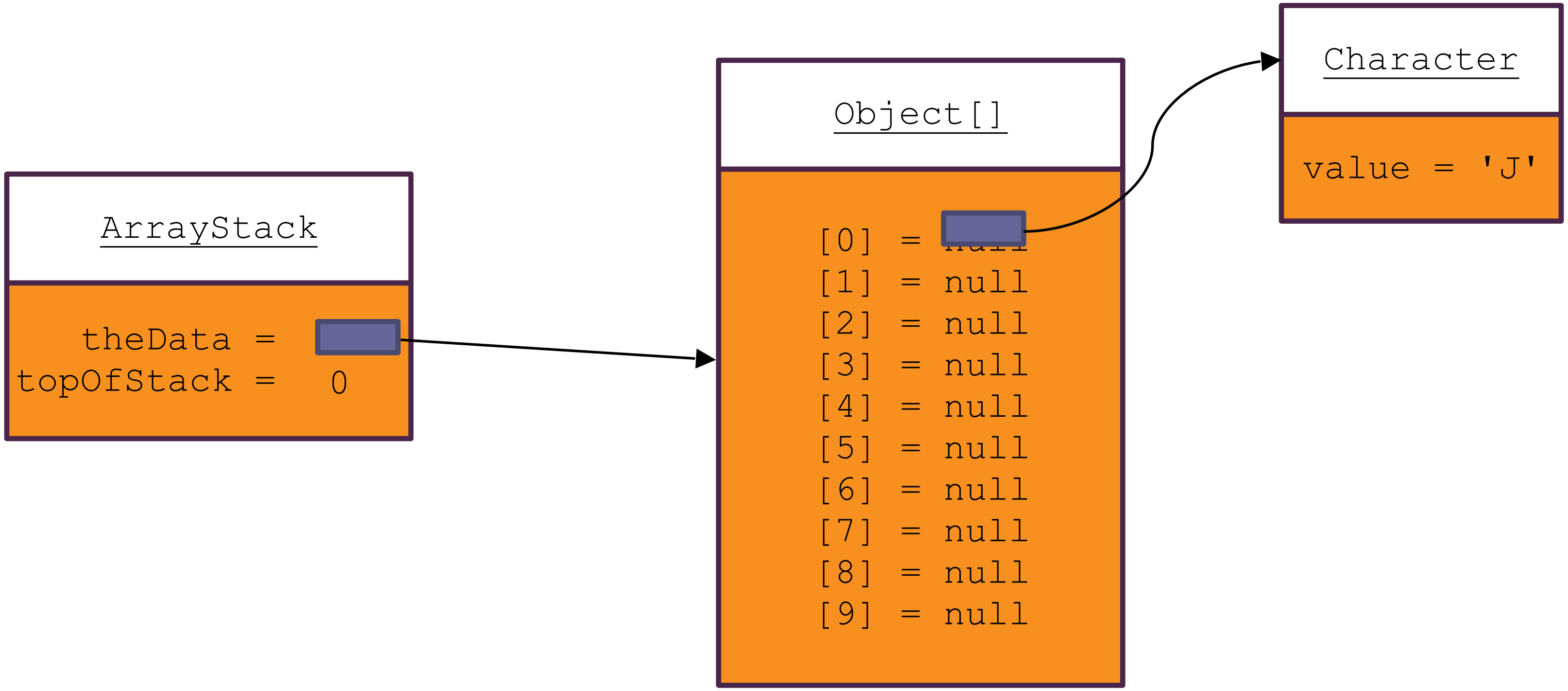
Object[]

[0] = null
[1] = null
[2] = null
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

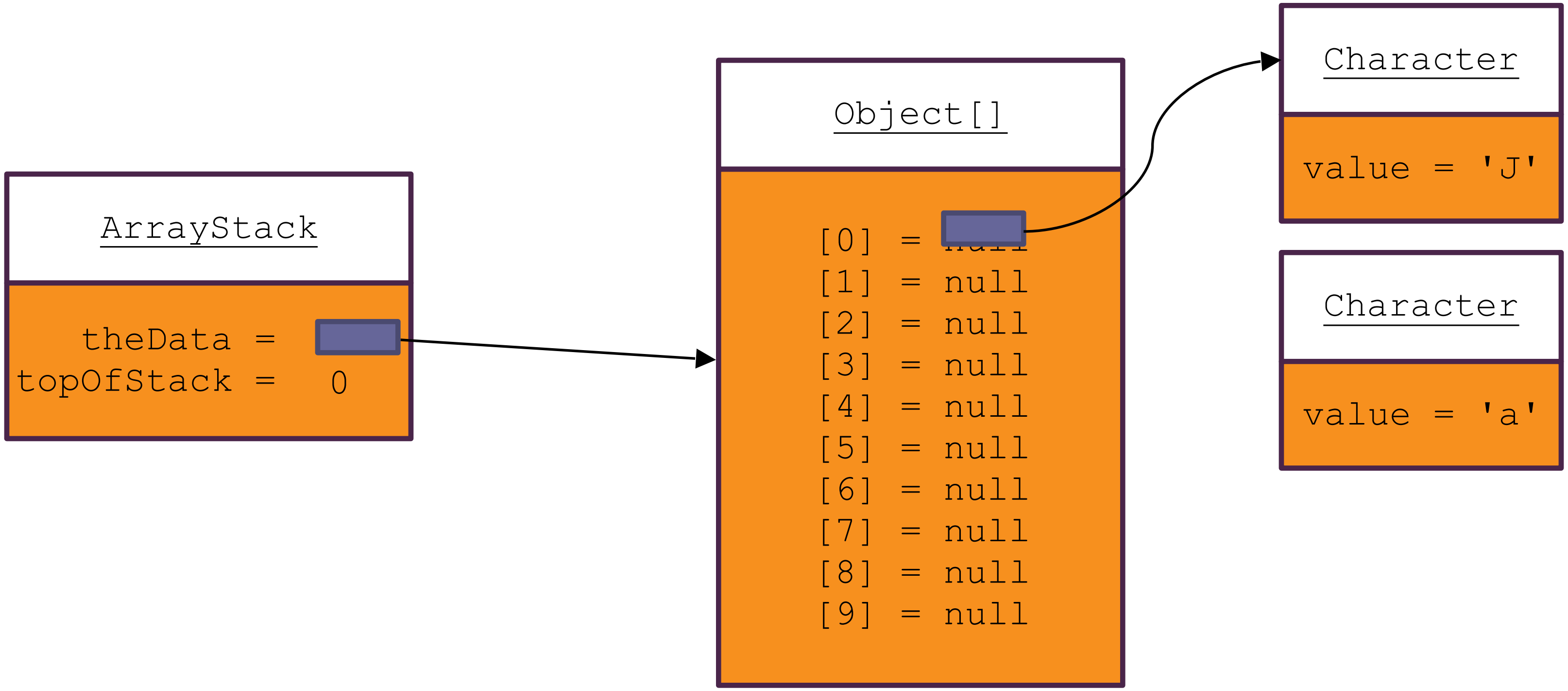
Character

value = 'J'

Implementing a Stack Using an Array



Implementing a Stack Using an Array



ArrayStack

theData =
topOfStack = 0

Object[]

[0] = null
[1] = null
[2] = null
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

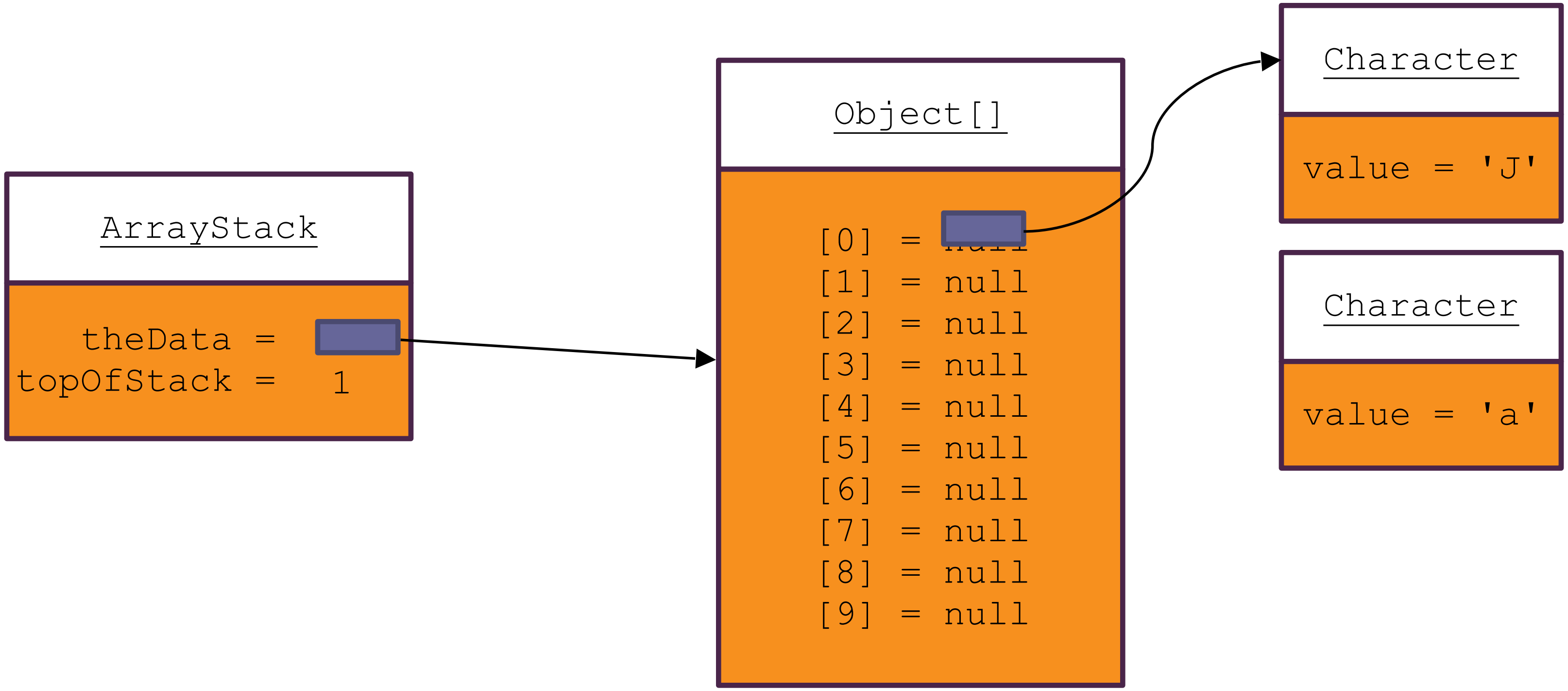
Character

value = 'J'

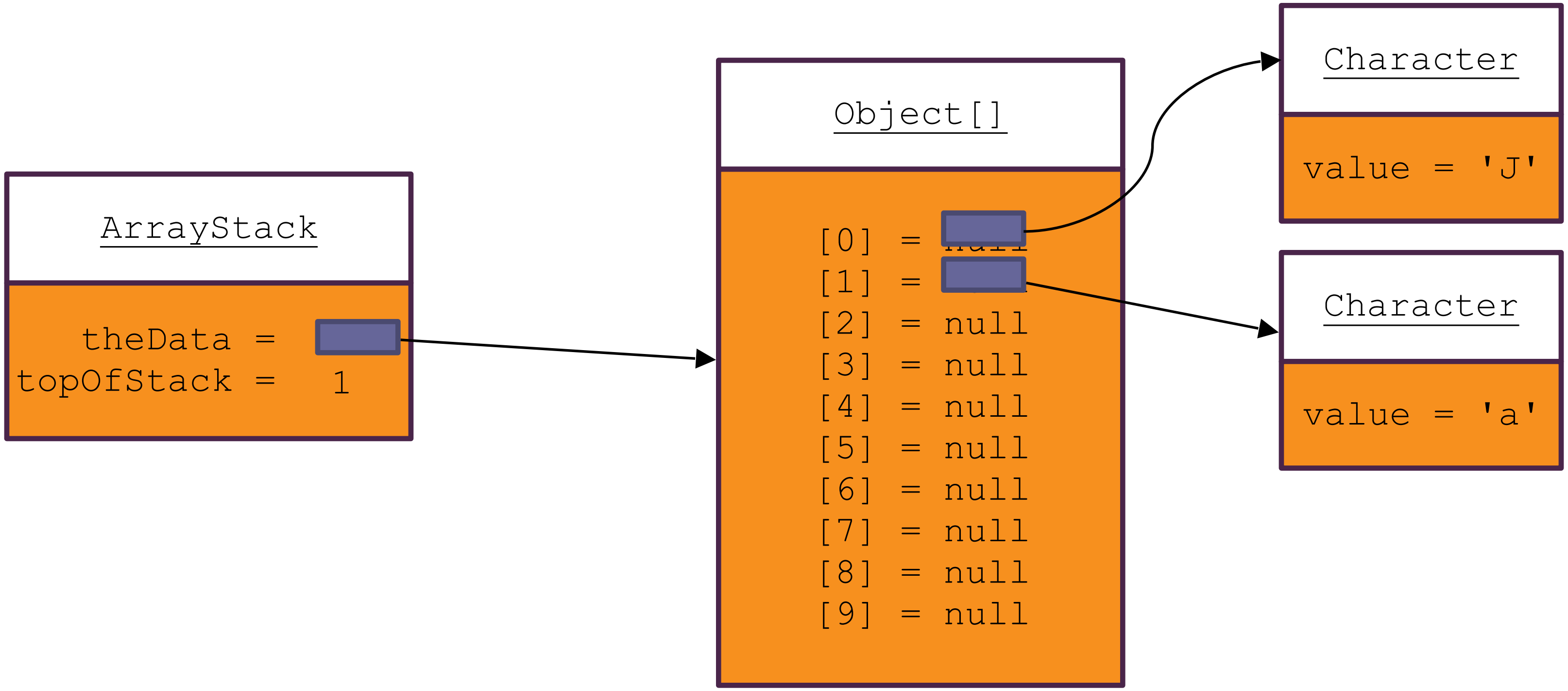
Character

value = 'a'

Implementing a Stack Using an Array



Implementing a Stack Using an Array



ArrayStack

theData =
topOfStack = 1

Object[]

[0] =
[1] =
[2] = null
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

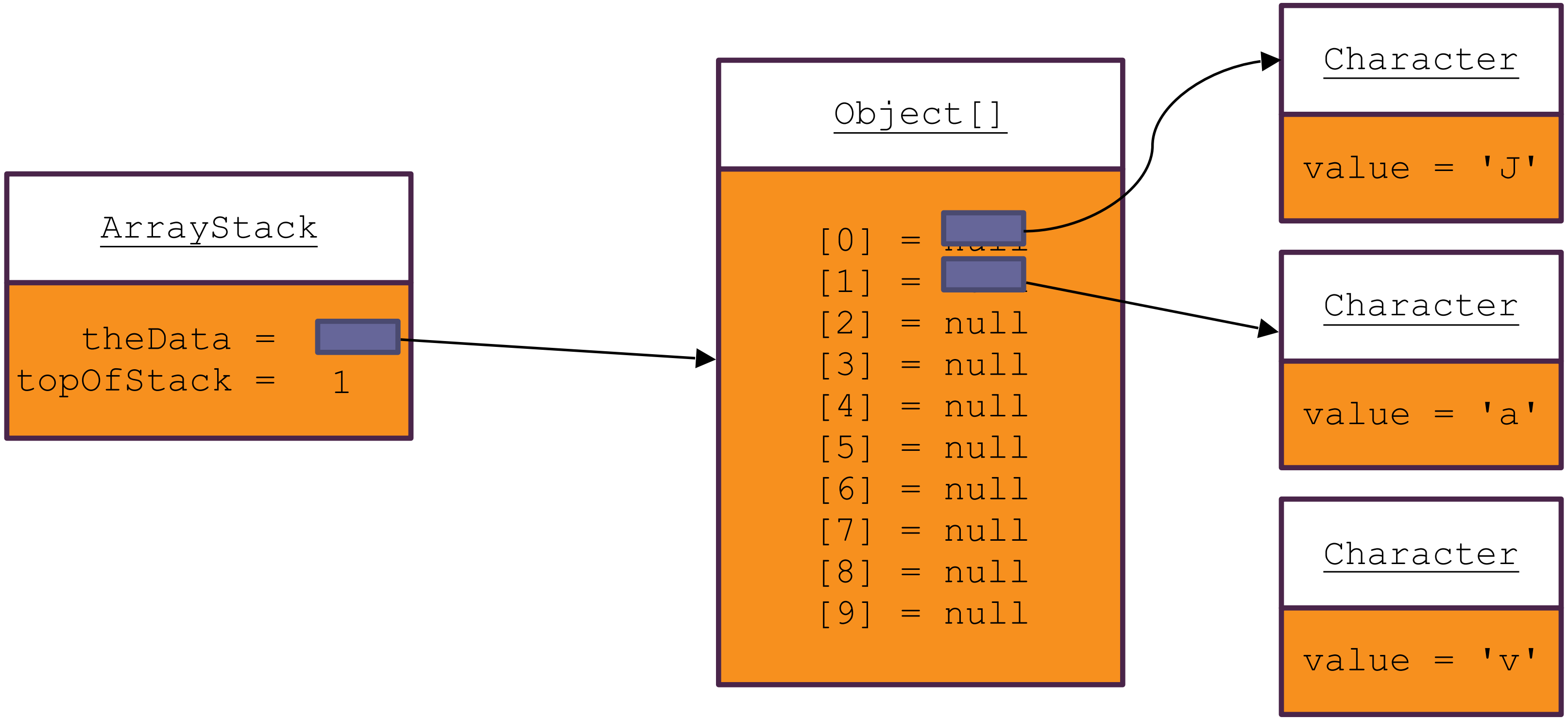
Character

value = 'J'

Character

value = 'a'

Implementing a Stack Using an Array



ArrayStack

theData =
topOfStack = 1

Object[]

[0] =
[1] =
[2] = null
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

Character

value = 'J'

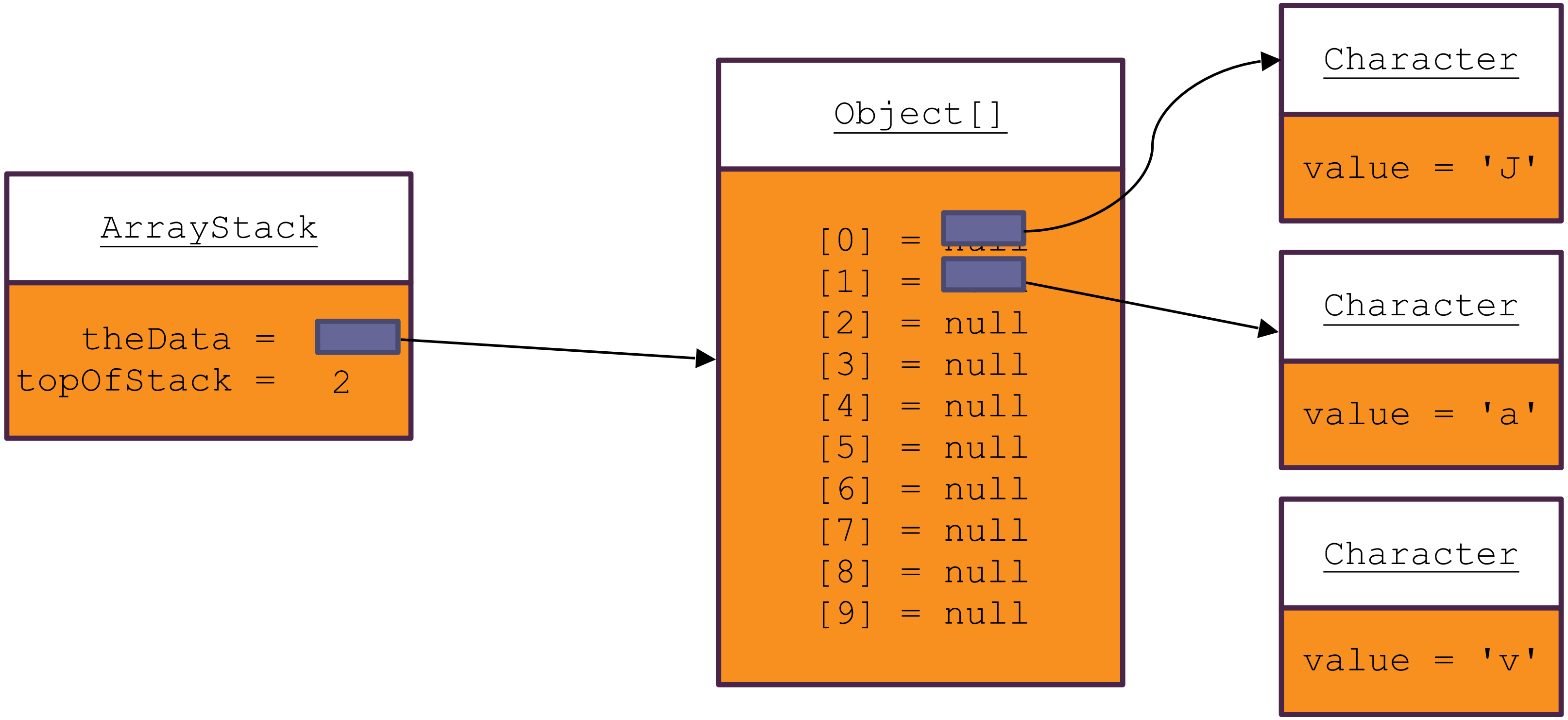
Character

value = 'a'

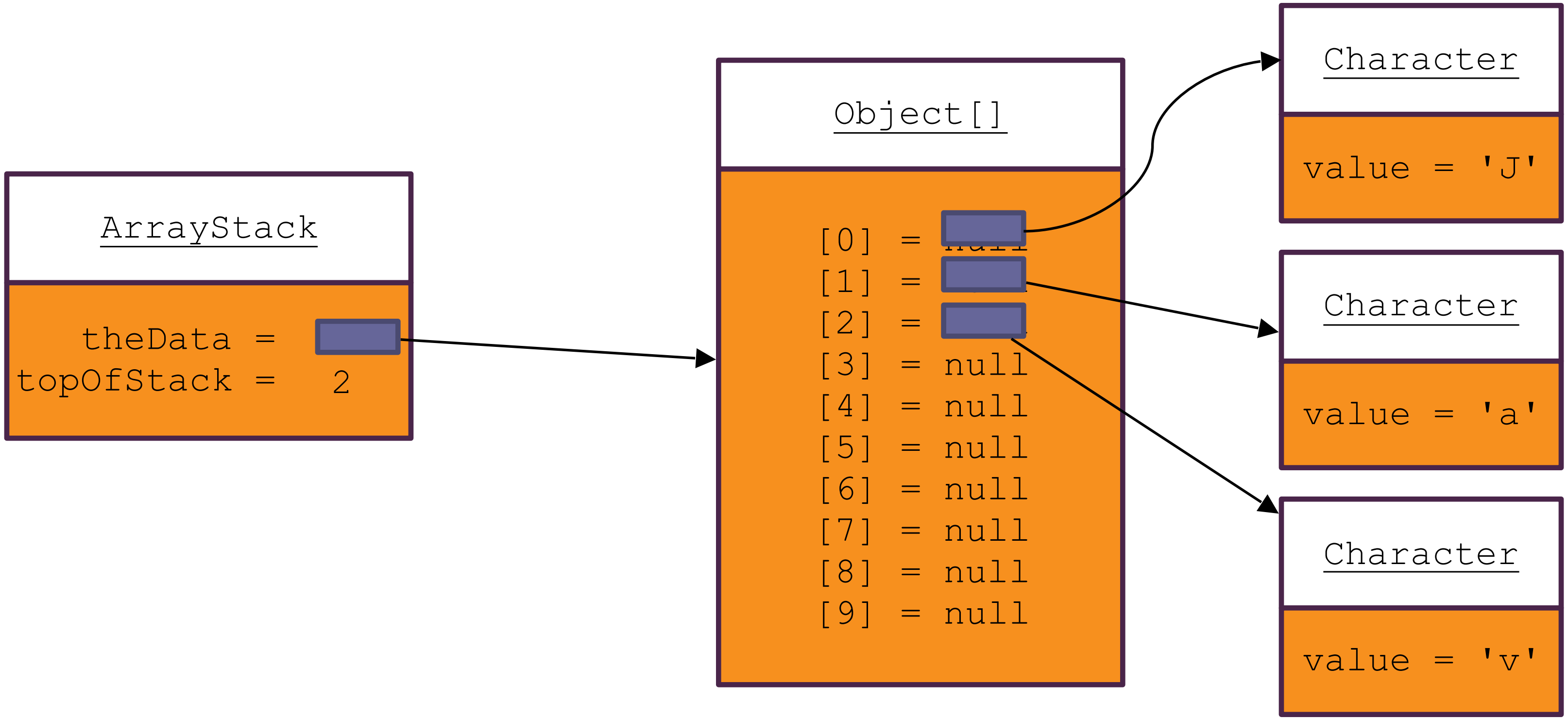
Character

value = 'v'

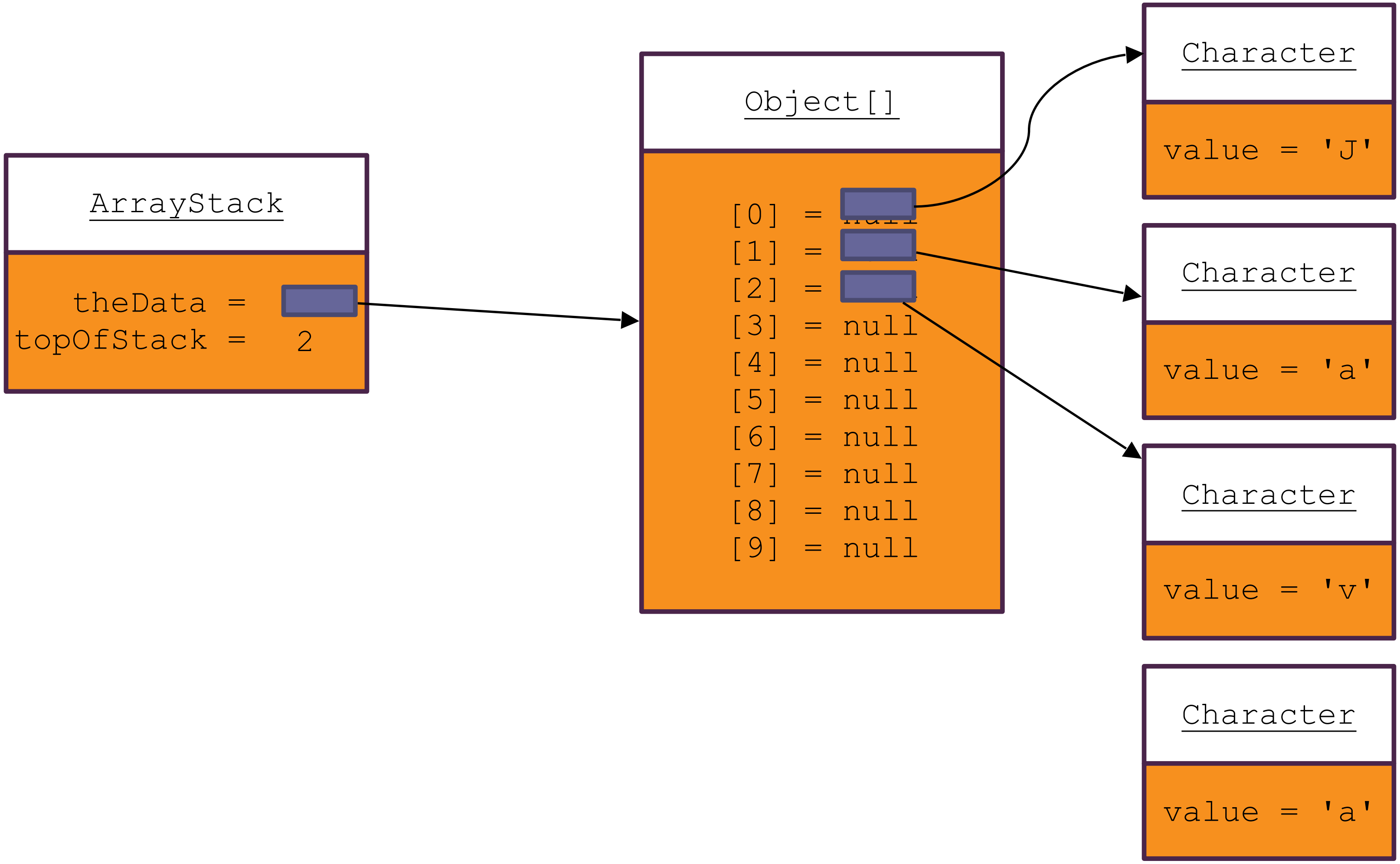
Implementing a Stack Using an Array



Implementing a Stack Using an Array



Implementing a Stack Using an Array



ArrayStack

theData =
topOfStack = 2

Object[]

[0] = null
[1] =
[2] =
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

Character

value = 'J'

Character

value = 'a'

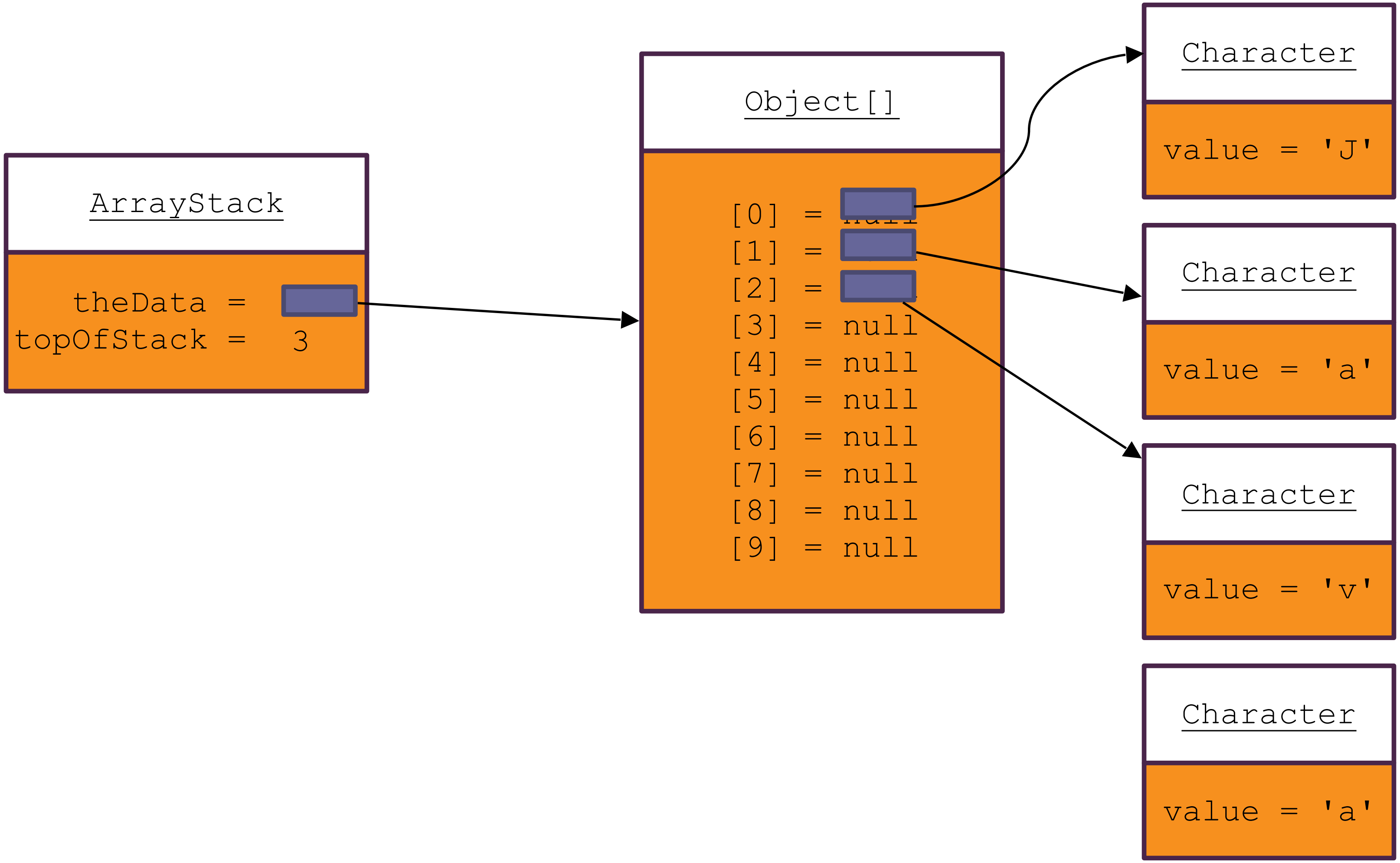
Character

value = 'v'

Character

value = 'a'

Implementing a Stack Using an Array



ArrayStack

theData =
topOfStack = 3

Object[]

[0] =
[1] =
[2] =
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

Character

value = 'J'

Character

value = 'a'

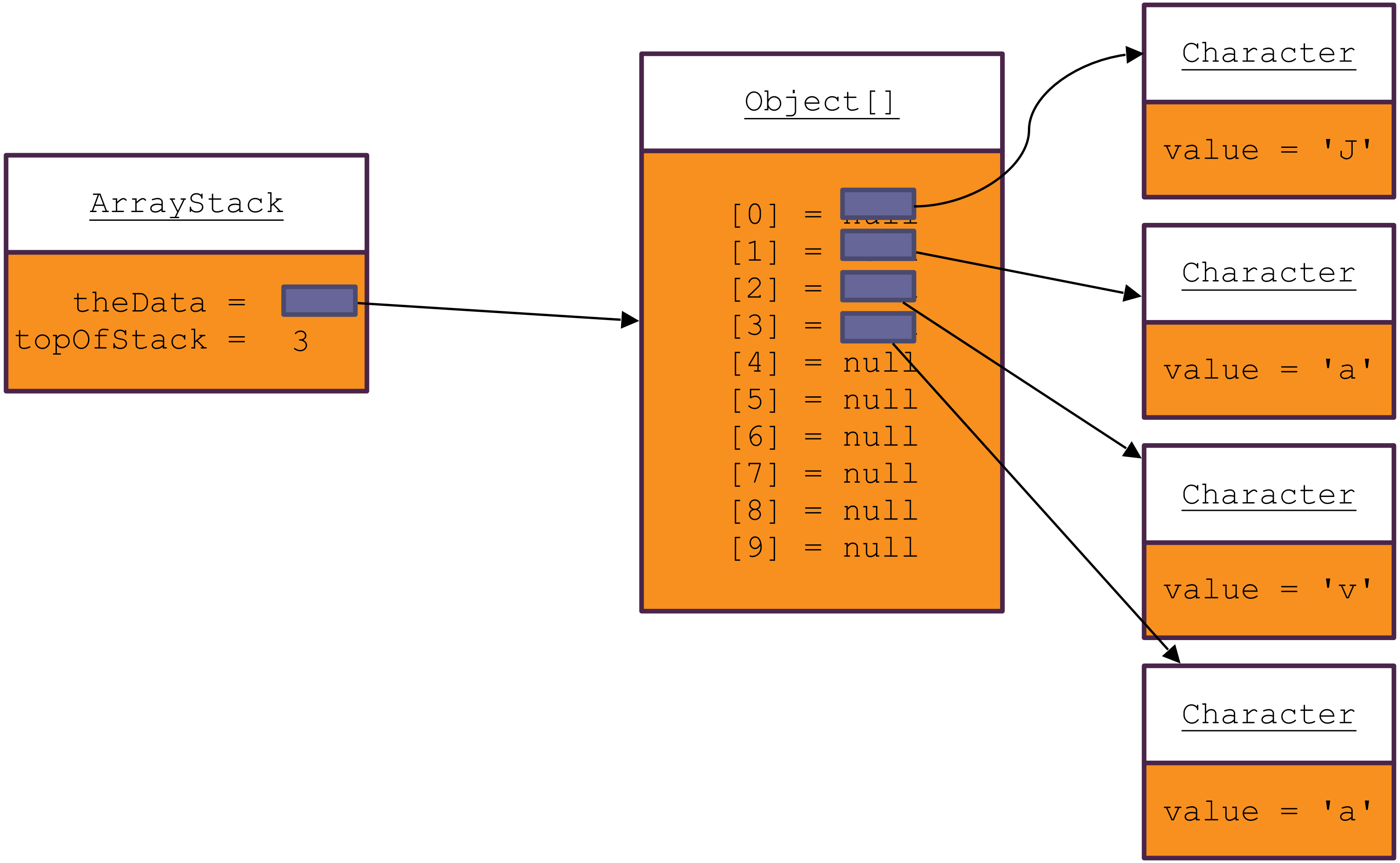
Character

value = 'v'

Character

value = 'a'

Implementing a Stack Using an Array



+ Comparison of java.util Stack Implementations

- Extending a `Vector` (as is done by Java) is a poor choice for stack implementation, since all `Vector` methods are accessible
- The easiest implementation adapts a `List` implementation for storing data (rather than extending List, just use one as a field in your Stack class.)
  - `ArrayList` is the favored by the book, but gives O(n) insertion
  - An underlying array requires reallocation of space when the array becomes full, and
  - an underlying linked data structure requires allocating storage for links
  - As all insertions and deletions occur at one end, they are (usually) constant time, O(1), regardless of the type of implementation used

Questions?

Testing