Introduction to Programming

03 Data Types

Stephan Krusche

15 November 2023
Technical University of Munich

Schedule

#	Date	Subject
1	18.10.23	Introduction
1b	25.10.23	Central exercise
	01.11.23	No lecture
2	08.11.23	Control Structures
3	**15.11.23**	**Data Types**
4	22.11.23	Object Orientation I
5	29.11.23	Object Orientation II
6	06.12.23	Object Orientation III
7	13.12.23	Algorithms
	20.12.23	No lecture
8	10.01.24	Programming Languages
9	17.01.24	Graphical User Interfaces
10	24.01.24	Recursion
11	31.01.24	Beyond Programming
12	07.02.24	Course Review

Roadmap of today’s lecture

- **Context**
  - You understand the basics of object oriented programming
  - You can use basic control structures (**`if`**, **`switch`**, **`for`**, **`while`**)
- **Learning goals**
  - Implement and use simple abstract data types such as **`List`**, **`Stack`** and **`Queue`**
  - Explain how **stacks** and **queues** work internally
  - Explain the differences between a **`List`** and an **`Array`**
  - Distinguish between **`List`** and **`Array`** based data types
  - Use simple built-in collection types such as **`List`** and **`Set`**
  - Iterate through lists and sets using the **`for each`** loop

Abstract data types

- Specify only the operations
- Hide details
    - Internal data structure
    - Implementation of the operations

→ Information Hiding

Motivation

- Preventing illegal access to the data structure
- Decouple sub-problems for
    - Implementation
    - Debugging
    - Maintenance
- Easy exchange of implementations (rapid prototyping)

Outline

➡ **List**

- Stack
- Queue

Java collection types

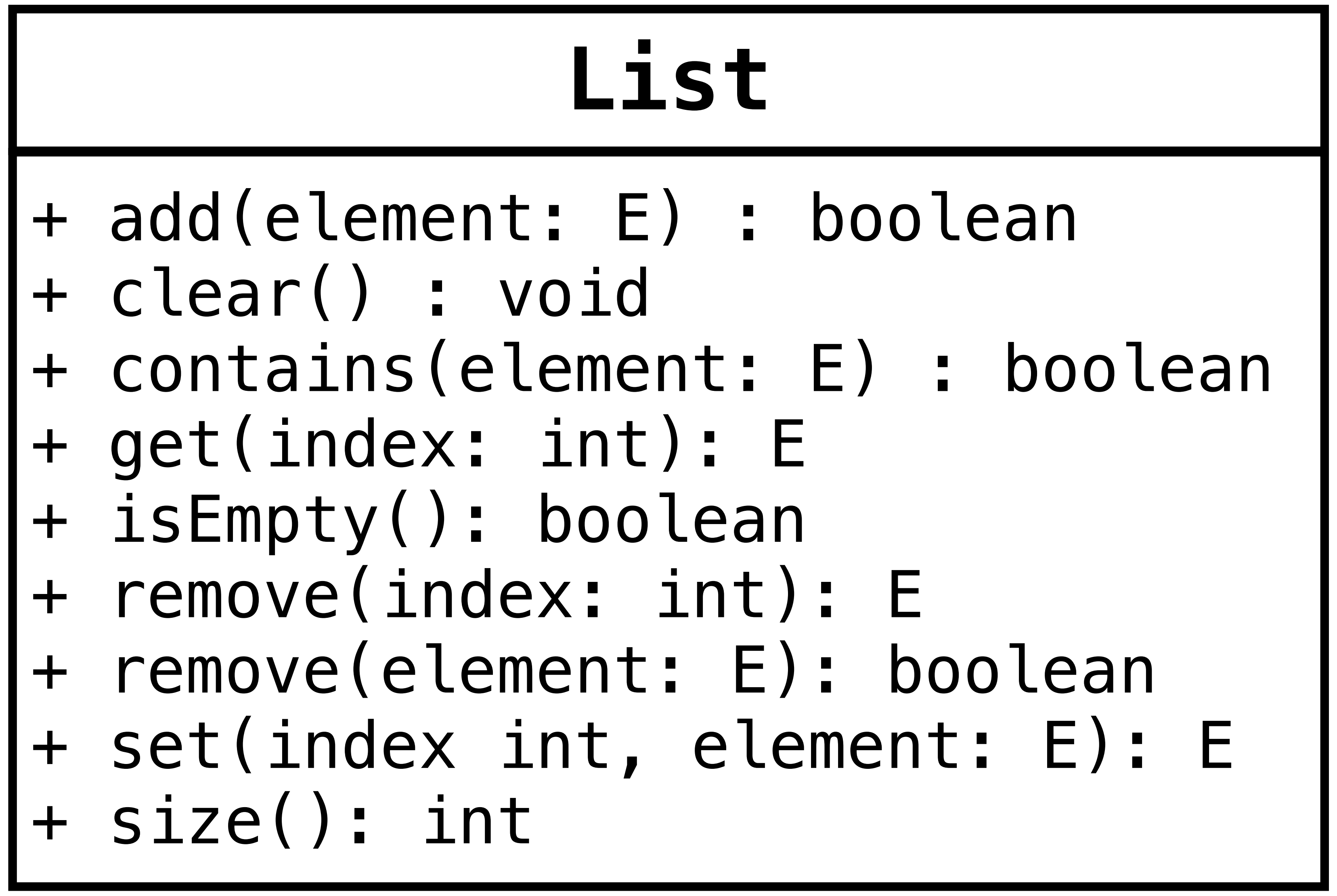
- Unified architecture for representation and manipulation of collections
- Reduction of programming effort
- Increase in the performance of operations on collections
- Interoperability of independent collections
- Most used types: **`List`**, **`Set`**, **`Queue`**, **`Map`**

http://download.oracle.com/javase/tutorial/collections/index.html

`java.util.List<E>`

Generic type

- The **`List`** **interface** represents an ordered sequence of objects



```
List
+ add(element: E) : boolean
+ clear() : void
+ contains(element: E) : boolean
+ get(index: int): E
+ isEmpty(): boolean
+ remove(index: int): E
+ remove(element: E): boolean
+ set(index int, element: E): E
+ size(): int
```

Excursion: generic types

More details in lecture 5

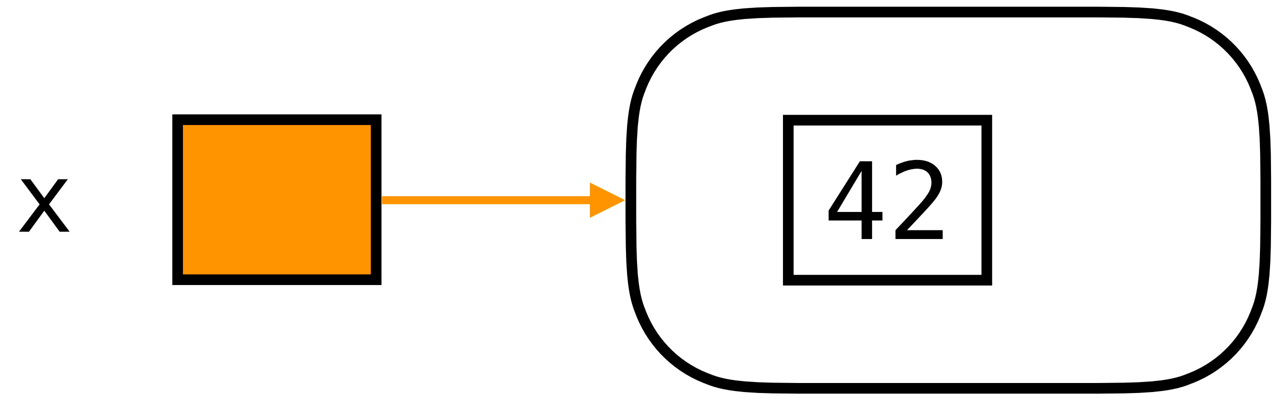
- Placeholder for actual types
- **Type safety**: ensure to use consistent data types
- **Reusability**: write code once and use it with different data types
- Prevent the need for casting and for exceptions —> the compiler can identify mistakes, e.g. adding an **`int`** into a **`String`** list
- Make the code cleaner and easier to understand

Wrapper classes for primitive types

Primitive type	**Wrapper class**
`byte`	**`Byte`**
`short`	**`Short`**
`int`	**`Integer`**
`long`	**`Long`**
`float`	**`Float`**
`double`	**`Double`**
`char`	**`Character`**
`boolean`	**`Boolean`**

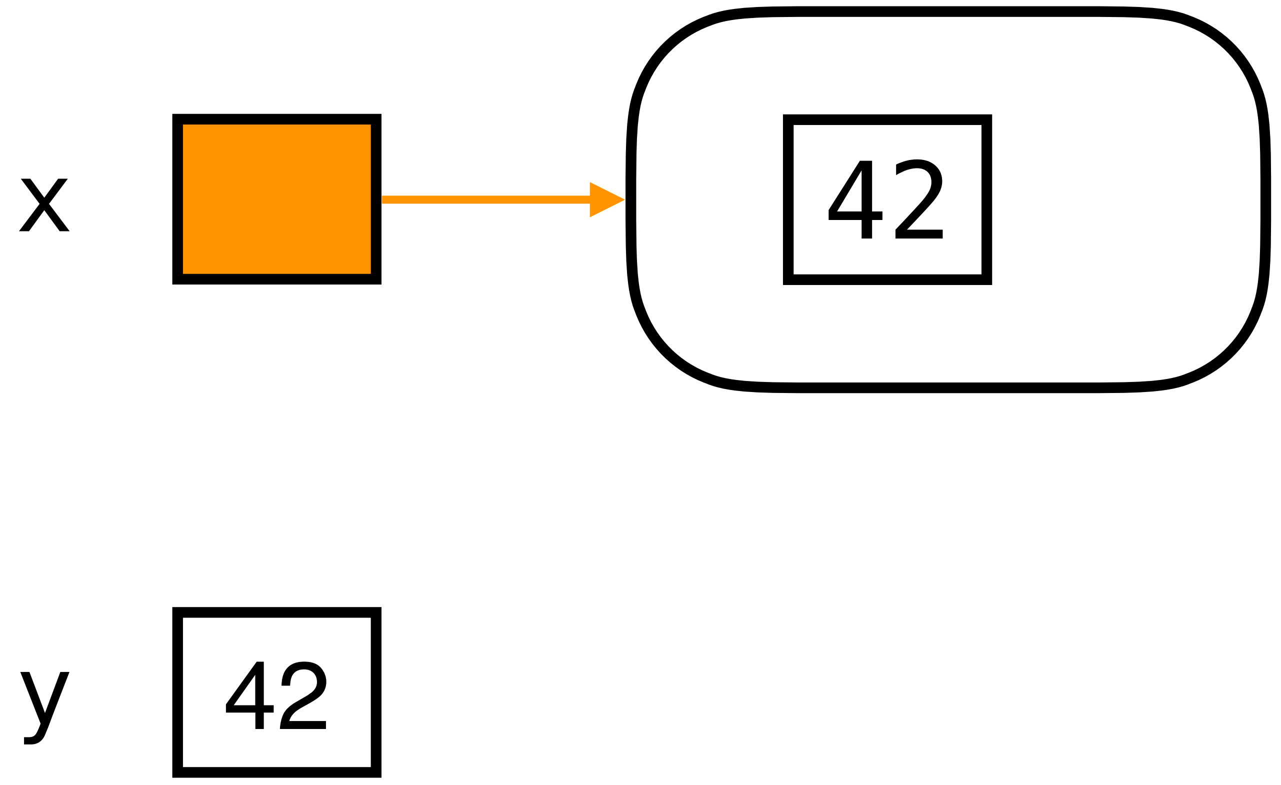
Example

```
Integer x = new Integer(42);
```



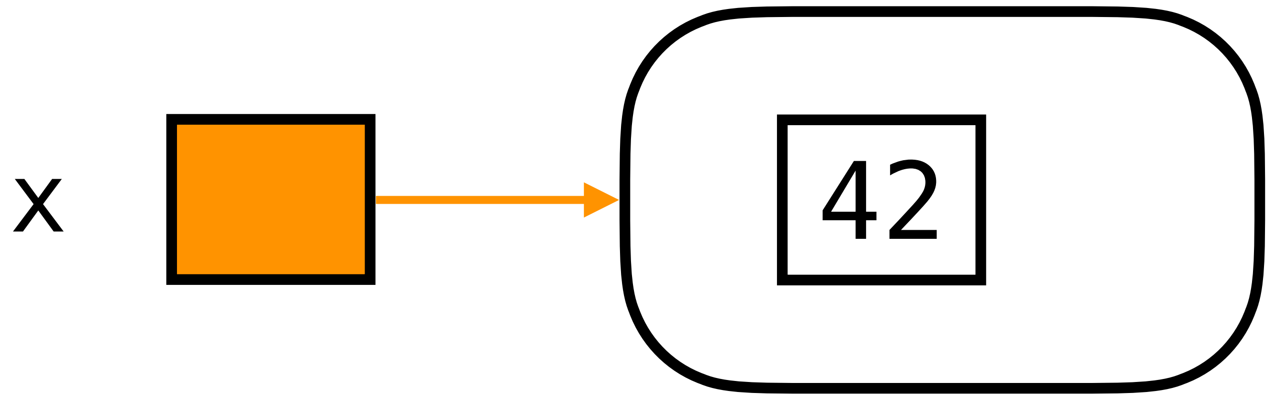
Unwrapping

- Wrapped values can also be unwrapped again
- Since Java 5 an assignment **`int y = x;`** is converted automatically

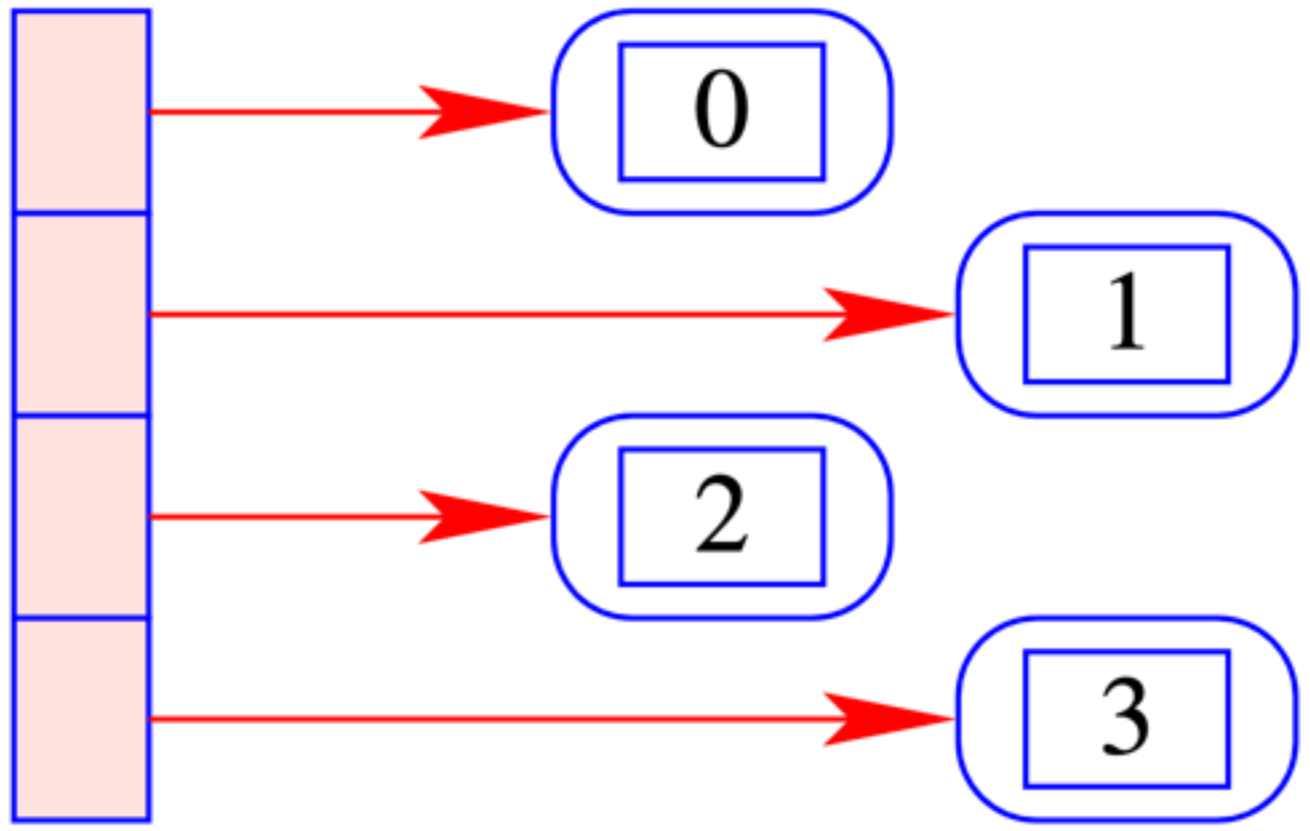


Wrapping

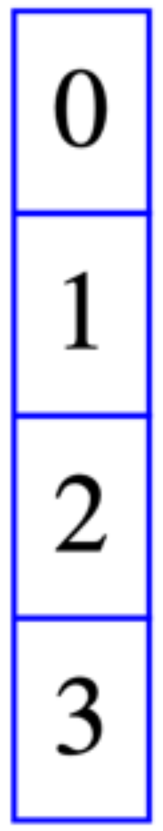
- Conversely, when an **`int`** value is assigned to an **`Integer`** variable, e.g. **`Integer x = 42;`** the constructor is called automatically



Integer vs. **int**



Integer []



int []

+ Integers can reside in collection data types with generic types (e.g. **List**, **Set**)
– They need more space (~ twice as much)
– They lead to many small objects distributed over the whole memory → worse cache behavior
– You have to deal with **null**

The class **Object**

Covered later

- The **`Object`** class is a common superclass for all classes
- All classes are therefore related to **`Object`**
- Useful methods of the **`Object`** class

```
String toString()
```

returns (any) representation as **`String`**

```
boolean equals(Object obj)
```

tests for object identity or reference equality

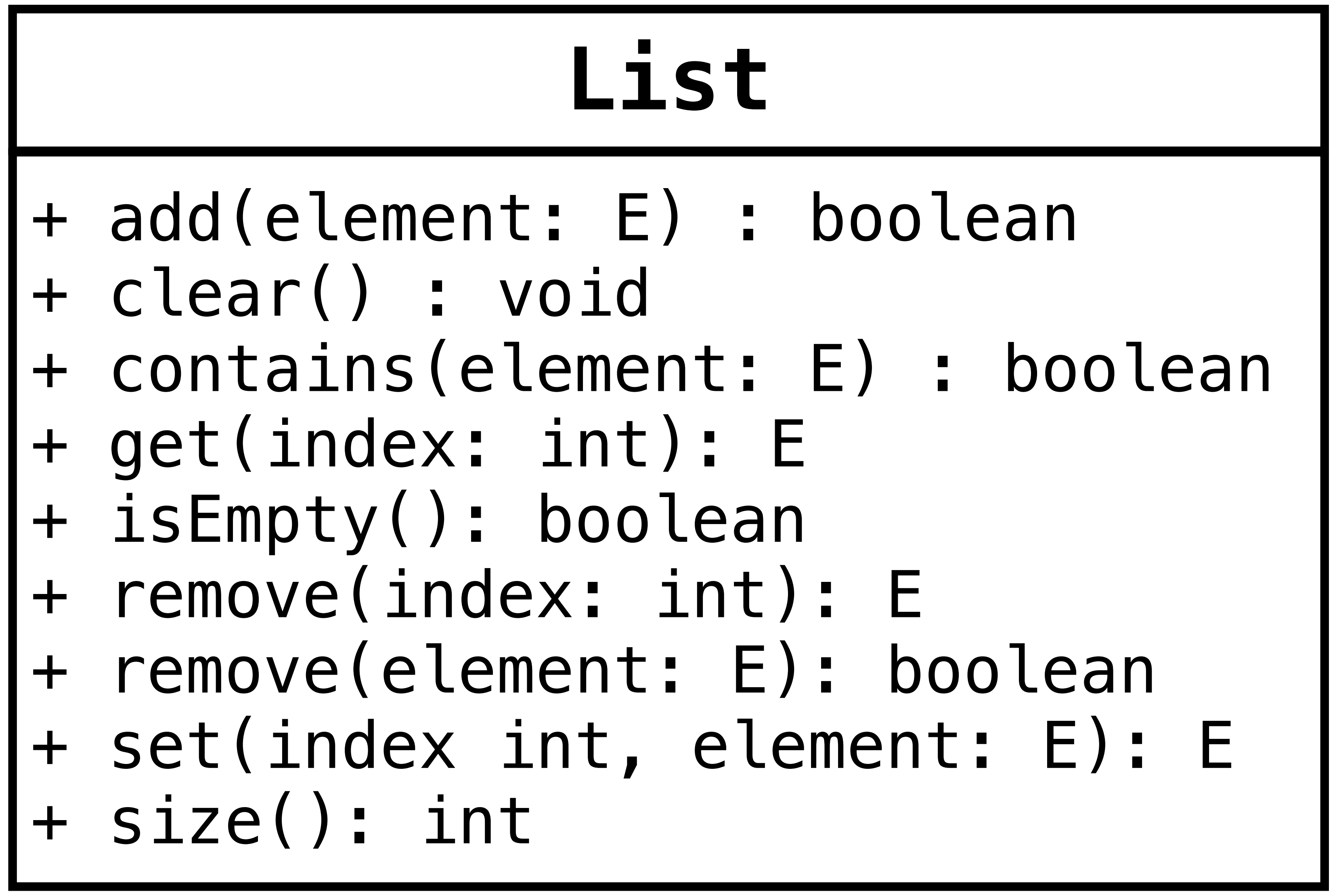
Example

```
public boolean equals(Object obj) {
    return this == obj;
}
```

`java.util.List<E>`

Generic type

- The **`List`** **interface** represents an ordered sequence of objects



```
List
+ add(element: E) : boolean
+ clear() : void
+ contains(element: E) : boolean
+ get(index: int): E
+ isEmpty(): boolean
+ remove(index: int): E
+ remove(element: E): boolean
+ set(index int, element: E): E
+ size(): int
```

Explanations

- **`add(element: E) : boolean`** - Adds an element to the end of the list and returns true if it was successful.
- **`clear() : void`** - Removes all elements from the list, leaving it empty.
- **`contains(element: E) : boolean`** - Checks if the list contains a specific element and returns true if it does.
- **`get(index: int): E`** - Retrieves the element at the specified position in the list.
- **`isEmpty(): boolean`** - Returns true if the list has no elements in it.
- **`remove(index: int): E`** - Removes and returns the element at the specified position in the list.
- **`remove(element: E): boolean`** - Removes the first occurrence of a specific element from the list, returning true if it was removed.
- **`set(index: int, element: E): E`** - Replaces the element at the specified position in the list with a new element, returning the original element.
- **`size(): int`** - Returns the number of elements currently in the list.

java.util.List<E>

Generic type

- Offers different implementations
- Example: ArrayList

```java
import java.util.*;

class Playground {
    public static void main(String[] args) {
        List<String> words = new ArrayList<String>();
        words.add("This");
        words.add("sentence");
        words.add("has");
        words.add("five");
        words.add("words");

        System.out.println(words.get(3));
        System.out.println(words.contains("sentence"));
        System.out.println(words.indexOf("This"));

        for (String word : words) {
            System.out.println(word);
        }
    }
}
```

for each loop

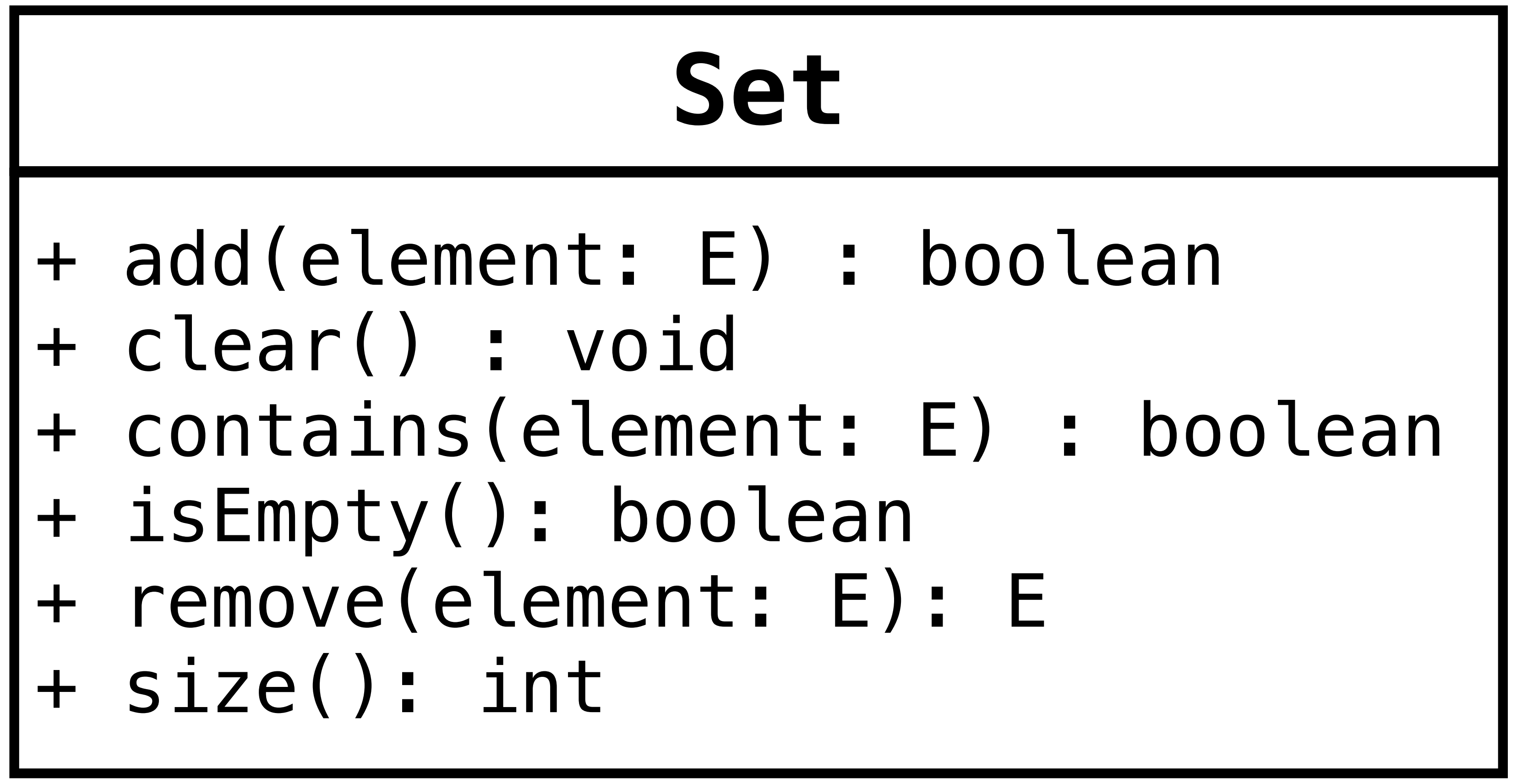
Output

```
five
true
0
This
sentence
has
five
words
```

java.util.Set<E>

Generic type

- The **Set** interface represents an unordered collection of objects that contains no duplicate elements (each element is unique)



```
Set
+ add(element: E) : boolean
+ clear() : void
+ contains(element: E) : boolean
+ isEmpty(): boolean
+ remove(element: E): E
+ size(): int
```

Explanations

- **`add(element: E) : boolean`** - Adds an element to the set and returns true if it was successful.
- **`clear() : void`** - Removes all elements from the set, leaving it empty.
- **`contains(element: E) : boolean`** - Checks if the set contains a specific element and returns true if it does.
- **`isEmpty(): boolean`** - Returns true if the set has no elements in it.
- **`remove(element: E): boolean`** - Removes a specific element from the set if it exists, returning true if it was removed.
- **`size(): int`** - Returns the number of elements currently in the list.

java.util.Set<E>

Generic type

- Offers different implementations
- Example: HashSet

```java
import java.util.*;

class Playground {
    public static void main(String[] args) {
        Set<String> words = new HashSet<String>();
        words.add("are");
        words.add("you");
        words.add("sure");
        words.add("you");
        words.add("are");
        words.add("right");

        System.out.println(words.size());

        for (String word : words) {
            System.out.println(word);
        }
    }
}
```

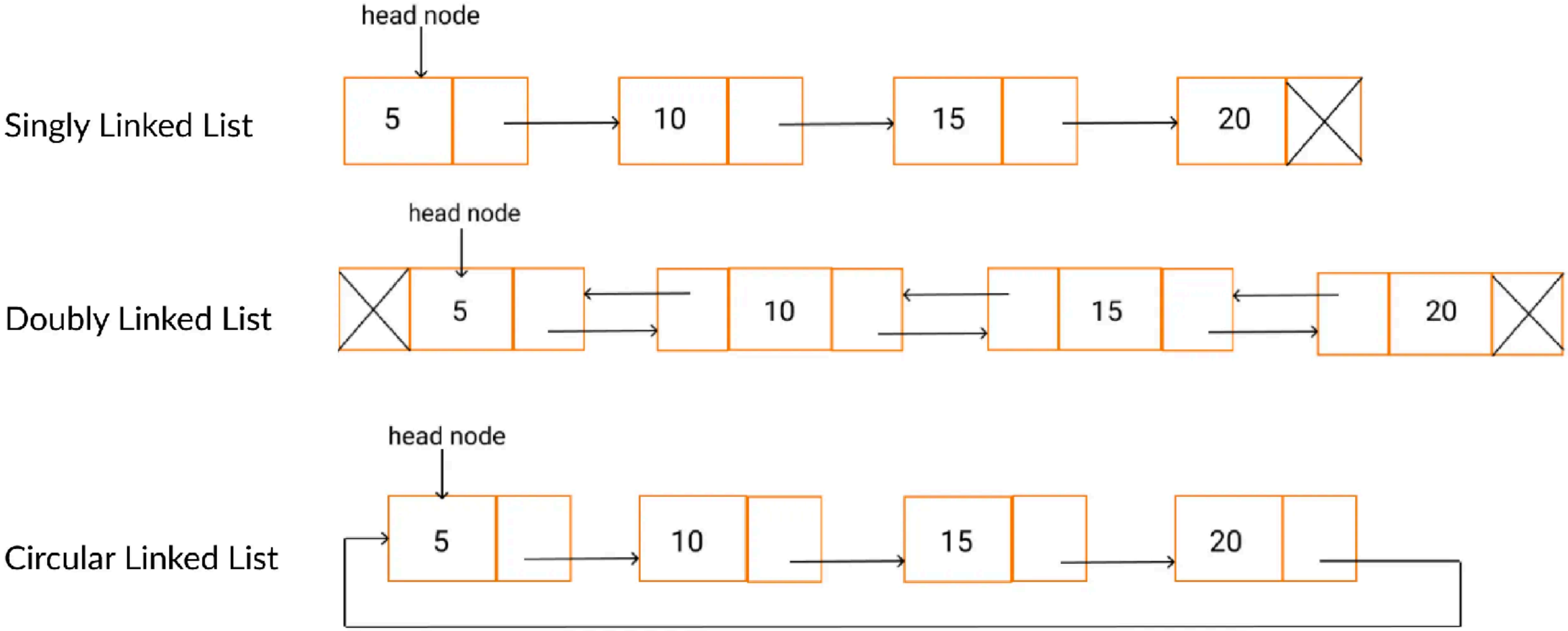
for each loop

Output

```
4
sure
are
right
you
```

No specific order

Linked lists



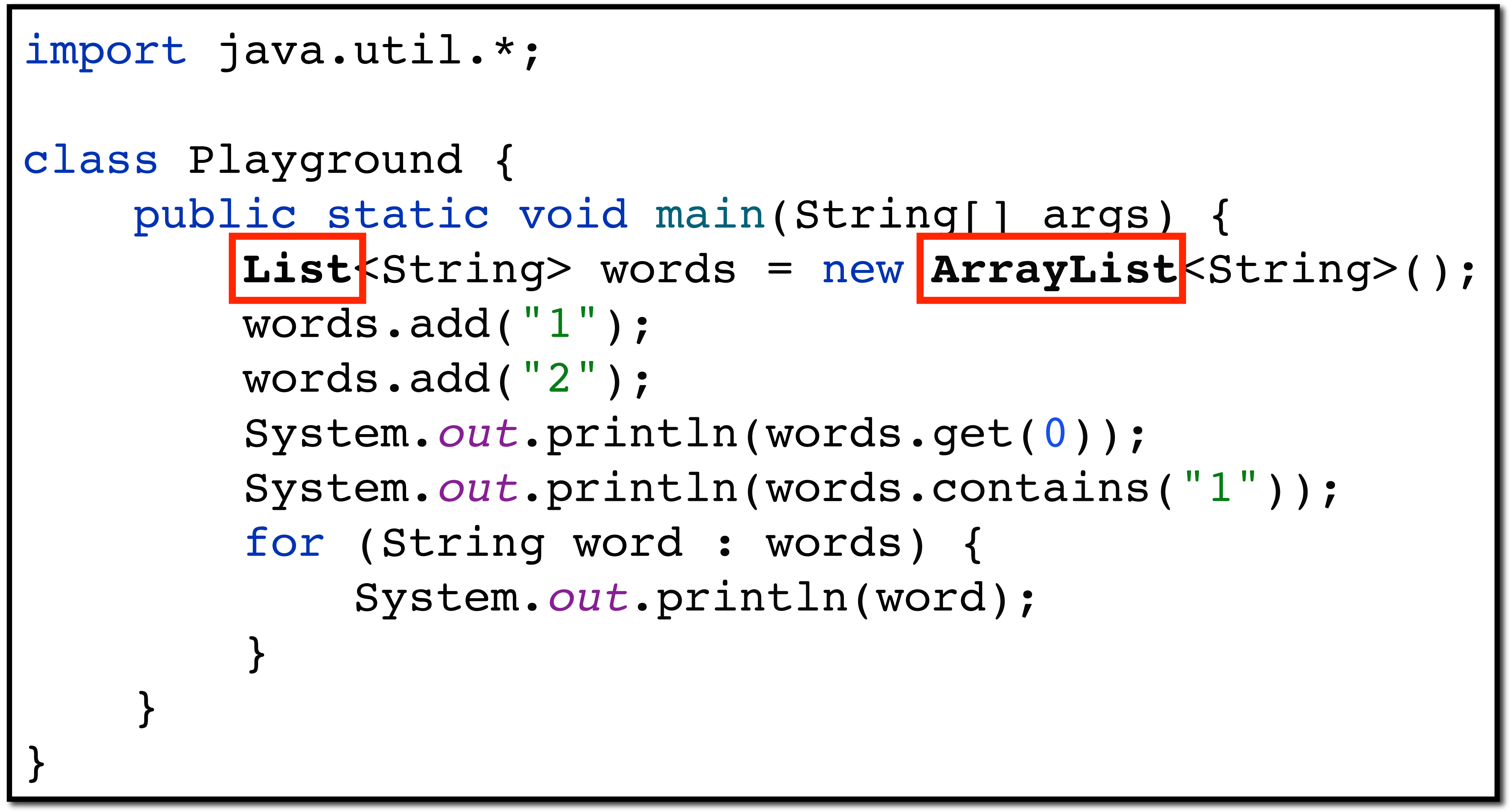
ArrayList vs LinkedList

- Both implementations in **java.util** follow the same interface

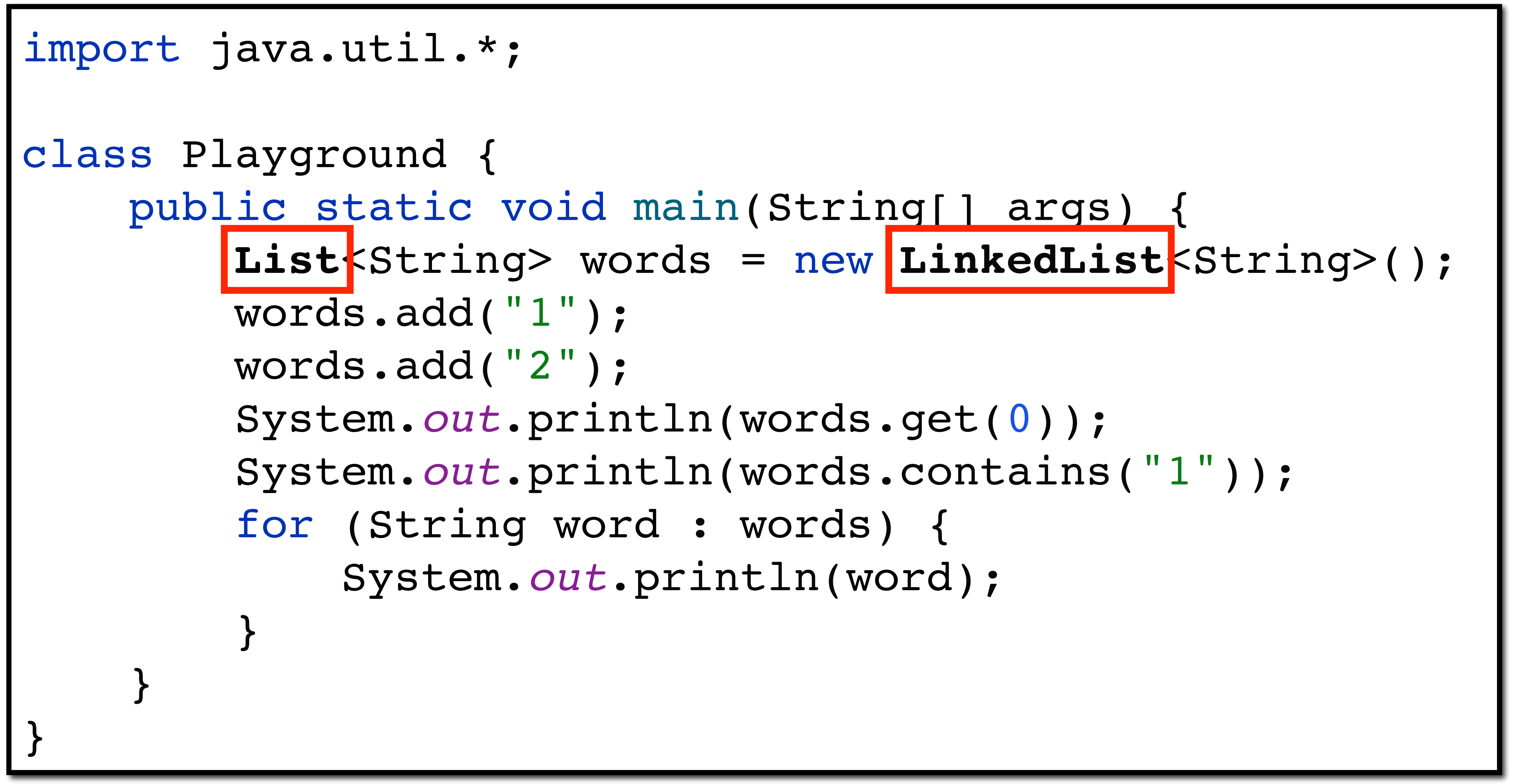
→ Same API (application programmer interface) and same methods

→ **Advantage**: both can be used interchangeably depending on memory and performance requirements

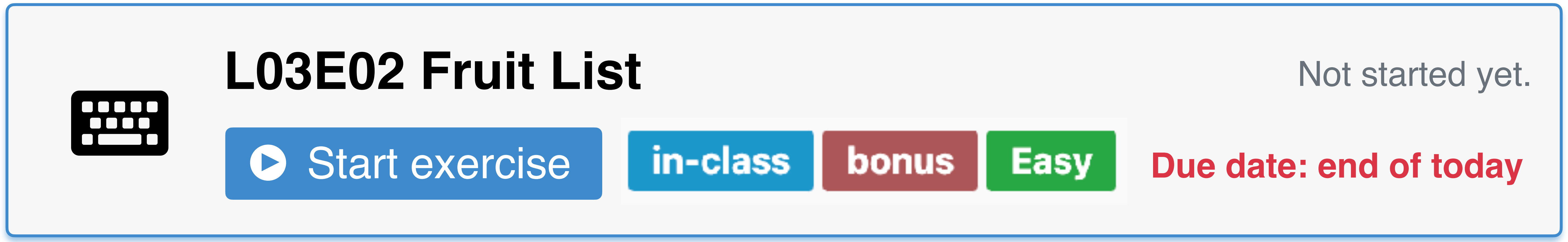
→ Changing the actual implementation is easy because only code places with the **new** operator must be changed



```
import java.util.*;

class Playground {
    public static void main(String[] args) {
        List<String> words = new ArrayList<String>();
        words.add("1");
        words.add("2");
        System.out.println(words.get(0));
        System.out.println(words.contains("1"));
        for (String word : words) {
            System.out.println(word);
        }
    }
}
```



```
import java.util.*;

class Playground {
    public static void main(String[] args) {
        List<String> words = new LinkedList<String>();
        words.add("1");
        words.add("2");
        System.out.println(words.get(0));
        System.out.println(words.contains("1"));
        for (String word : words) {
            System.out.println(word);
        }
    }
}
```



L03E02 Fruit List

Not started yet.

Due date: end of today

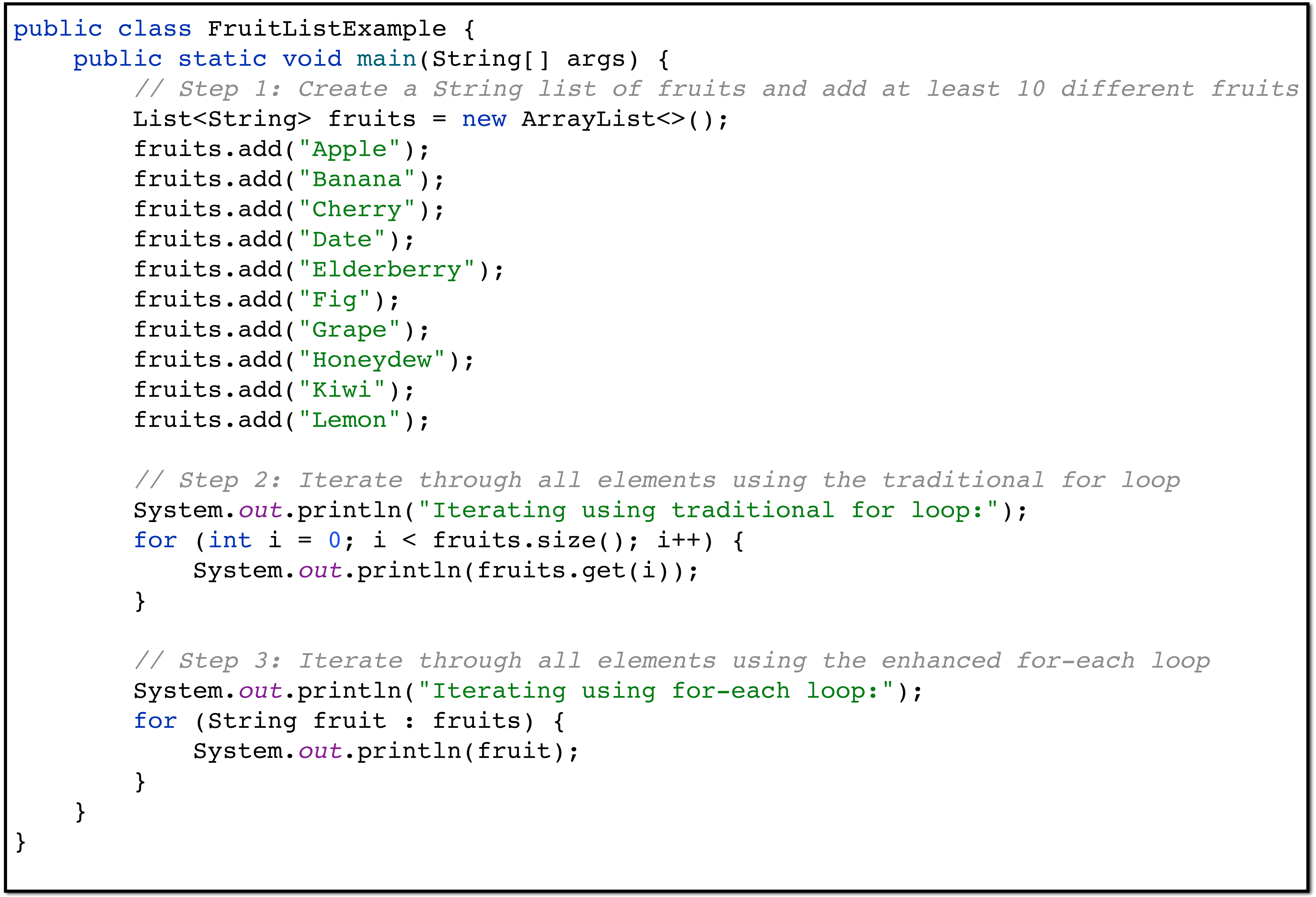
10 min

4 pts

- **Problem**: manipulate fruits in a list

1. Create a **`String`** list **`fruits`** and add at least 10 different fruits
2. Iterate through all elements using the “traditional” for loop
   Hint: **`for(int i = 0; i < fruits.size(); i++)`**
3. Iterate through all elements using the enhanced for each loop
   Hint: **`for(String fruit : fruits)`**
4. Remove all fruits that contain the letter “e”

Example solution



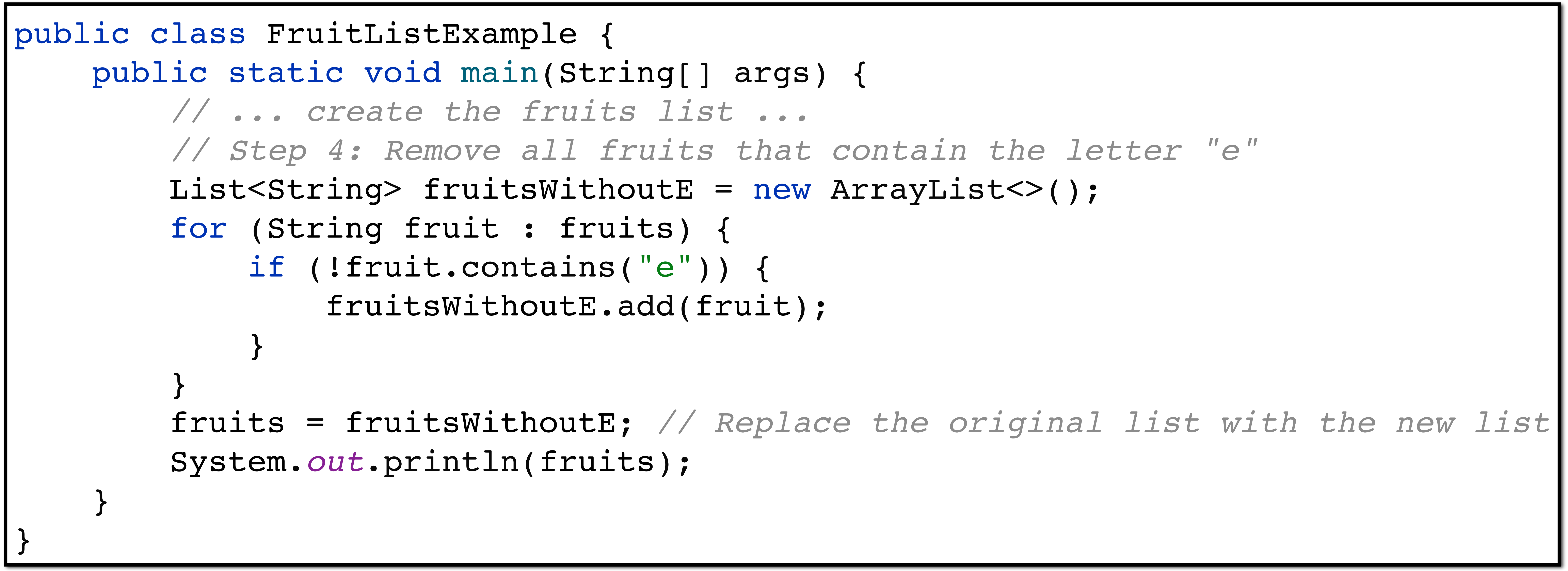
```java
public class FruitListExample {
    public static void main(String[] args) {
        // Step 1: Create a String list of fruits and add at least 10 different fruits
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Date");
        fruits.add("Elderberry");
        fruits.add("Fig");
        fruits.add("Grape");
        fruits.add("Honeydew");
        fruits.add("Kiwi");
        fruits.add("Lemon");

        // Step 2: Iterate through all elements using the traditional for loop
        System.out.println("Iterating using traditional for loop:");
        for (int i = 0; i < fruits.size(); i++) {
            System.out.println(fruits.get(i));
        }

        // Step 3: Iterate through all elements using the enhanced for-each loop
        System.out.println("Iterating using for-each loop:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

~~Example solution~~

```java
public class FruitListExample {
    public static void main(String[] args) {
        // ... create the fruits list ...
        // Step 4: Remove all fruits that contain the letter "e"
        System.out.println("Removing fruits containing 'e':");
        for (String fruit : fruits) {
            if (fruit.contains("e")) {
                fruits.remove(fruit);
            }
        }
        System.out.println(fruits);
    }
}
```

Note: this will lead to a **`ConcurrentModificationException`**

Example solution



```java
public class FruitListExample {
    public static void main(String[] args) {
        // ... create the fruits list ...
        // Step 4: Remove all fruits that contain the letter "e"
        List<String> fruitsWithoutE = new ArrayList<>();
        for (String fruit : fruits) {
            if (!fruit.contains("e")) {
                fruitsWithoutE.add(fruit);
            }
        }
        fruits = fruitsWithoutE; // Replace the original list with the new list
        System.out.println(fruits);
    }
}
```

Error handling and common pitfalls

- Collections (**`Set`**, **`List`**, etc.) contain object data types and therefore can contain **`null`** values
  - To be safe, check for **`null`** when retrieving single elements
- Concurrent modification exceptions occur when a collection is modified while iterating over
  - Use workarounds to modify the collection after the iteration, e.g. collecting elements to be removed in a second collection
- Some operations in collections are inefficient (e.g. adding an element at the beginning of a list) and should be avoided if possible

Break

10 min

The lecture will continue at **15:05**

Outline

- List
- ➡ **Stack**
- Queue

Stack

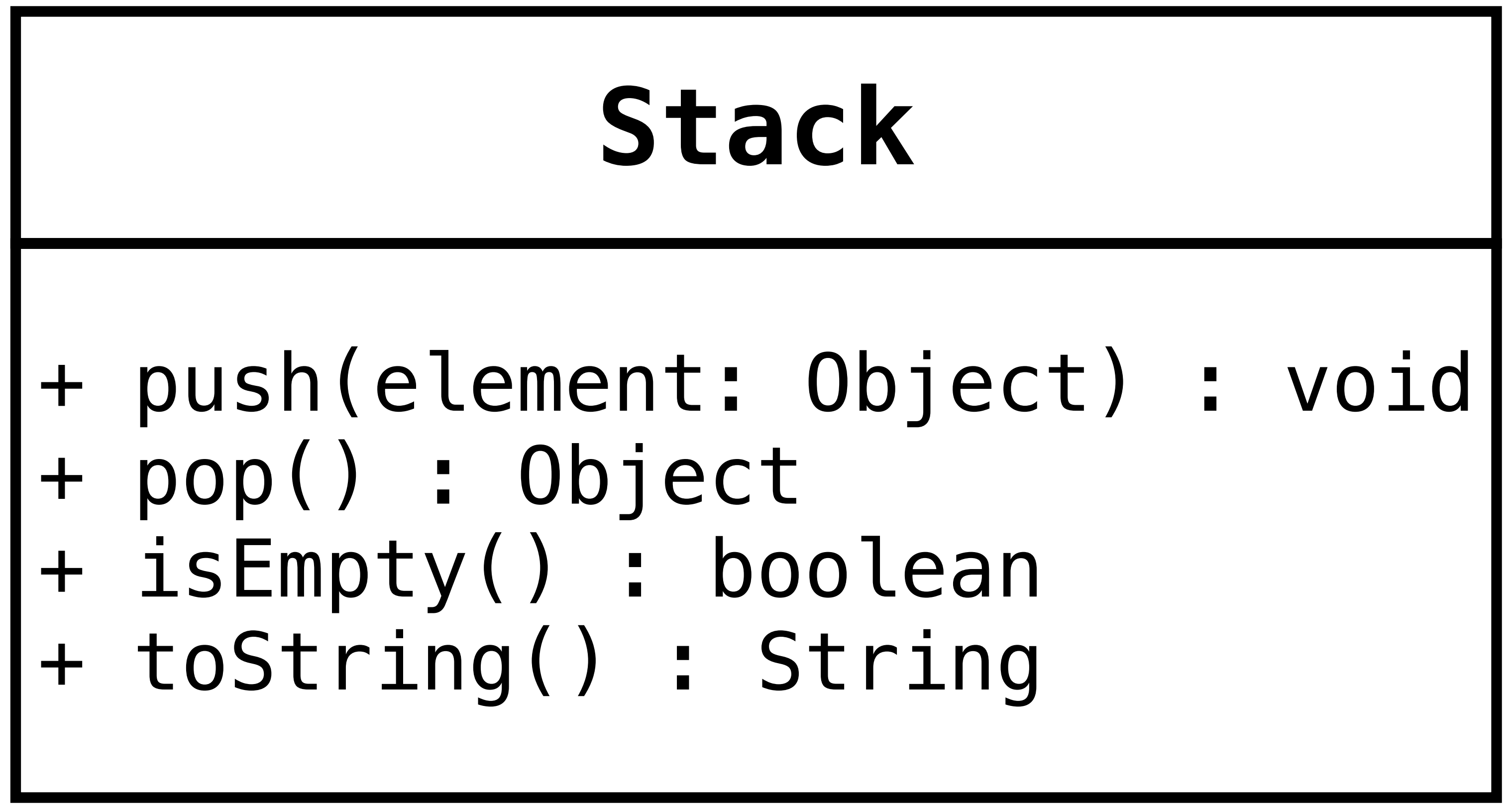
- Operations

`void push(Object element)`	puts **`element`** on top of the stack
`Object pop()`	returns top **`element`**
`boolean isEmpty()`	tests for emptiness
`String toString()`	returns a string representation

- Follows the LIFO principle: last in, first out

Friedrich Ludwig Bauer, TUM

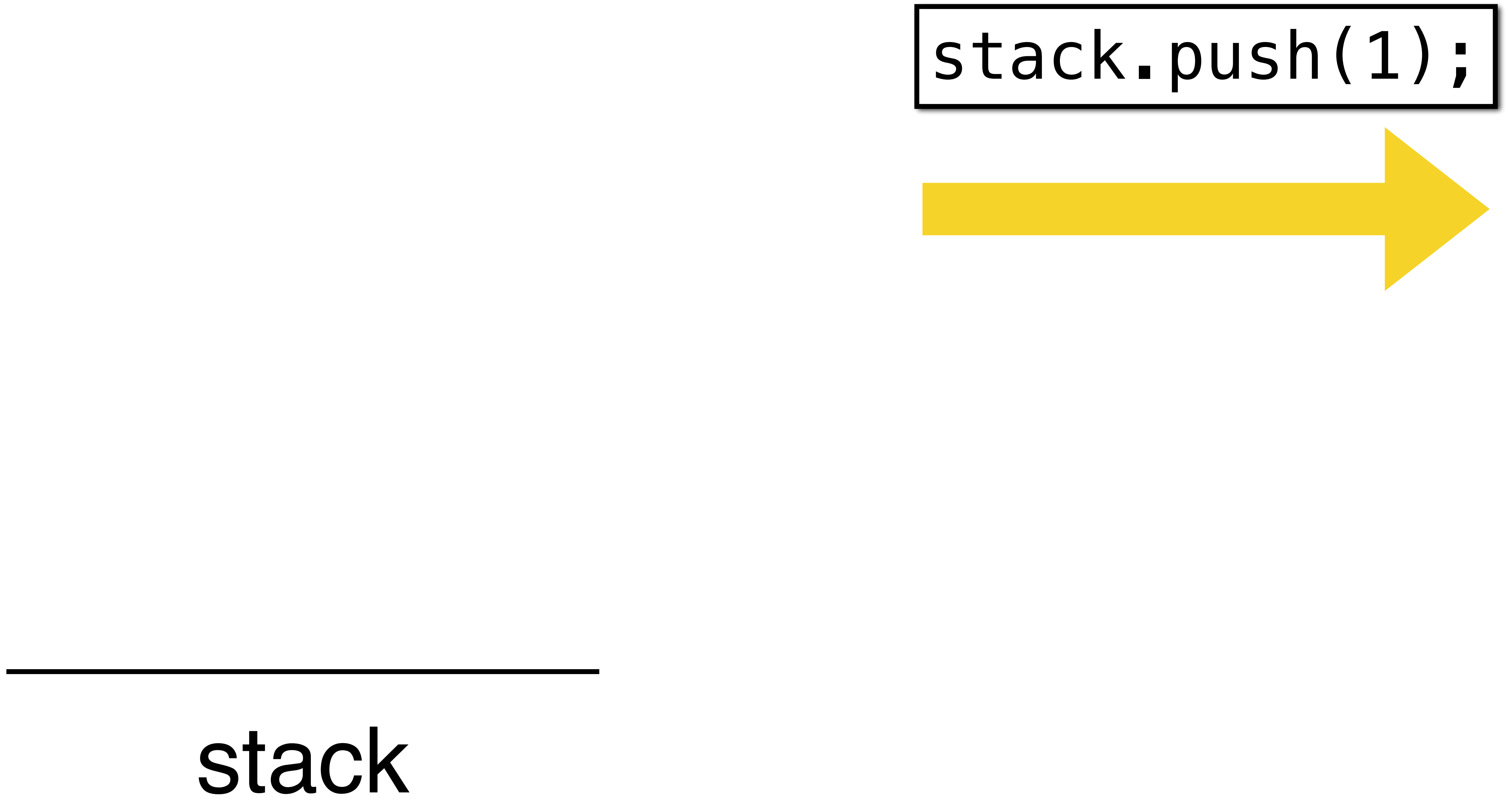
Modeling a stack



Stack

```
+ push(element: Object) : void
+ pop() : Object
+ isEmpty() : boolean
+ toString() : String
```

Stack visualization

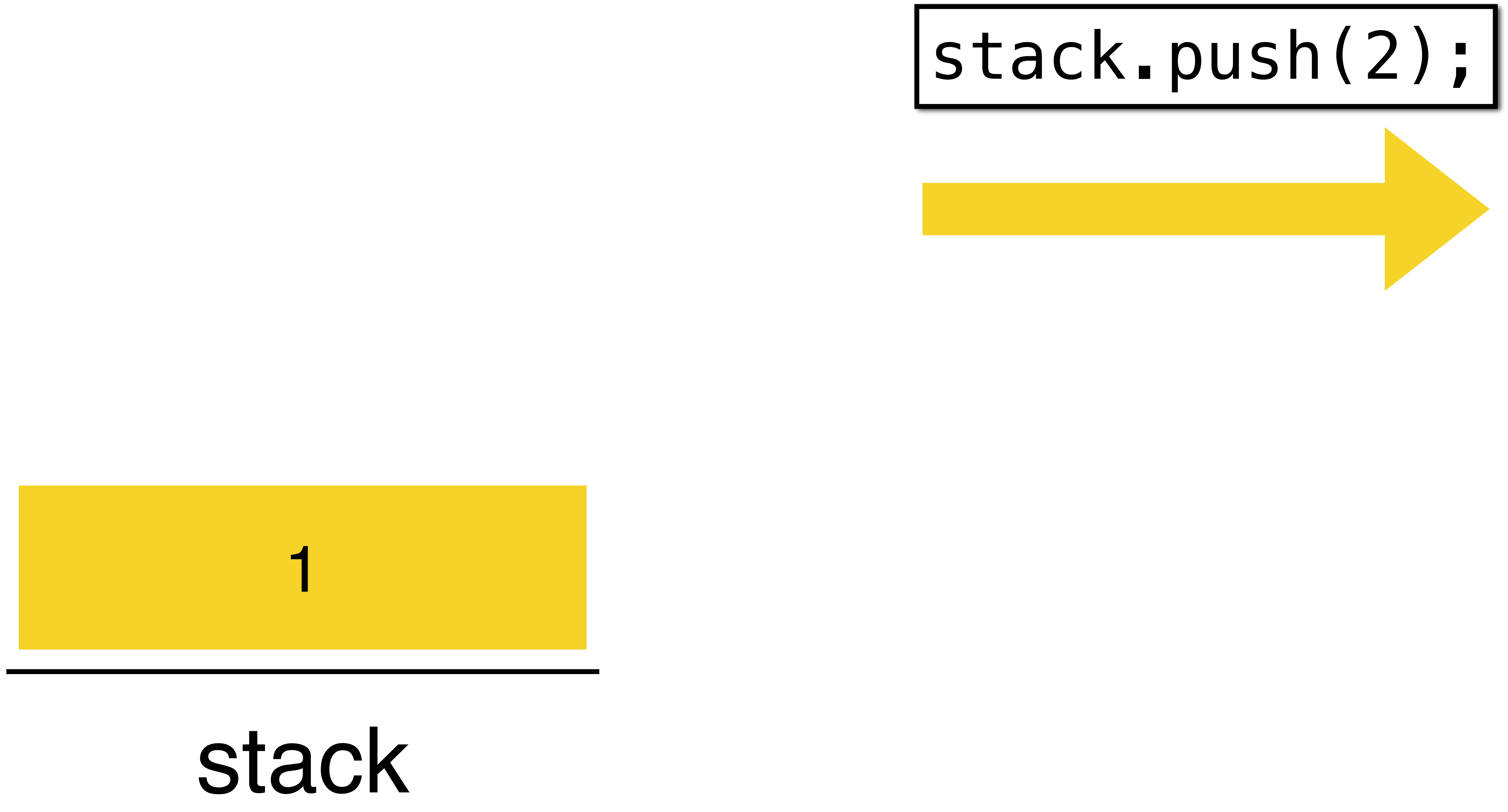


```
stack.push(1);
```

Size: 0

stack

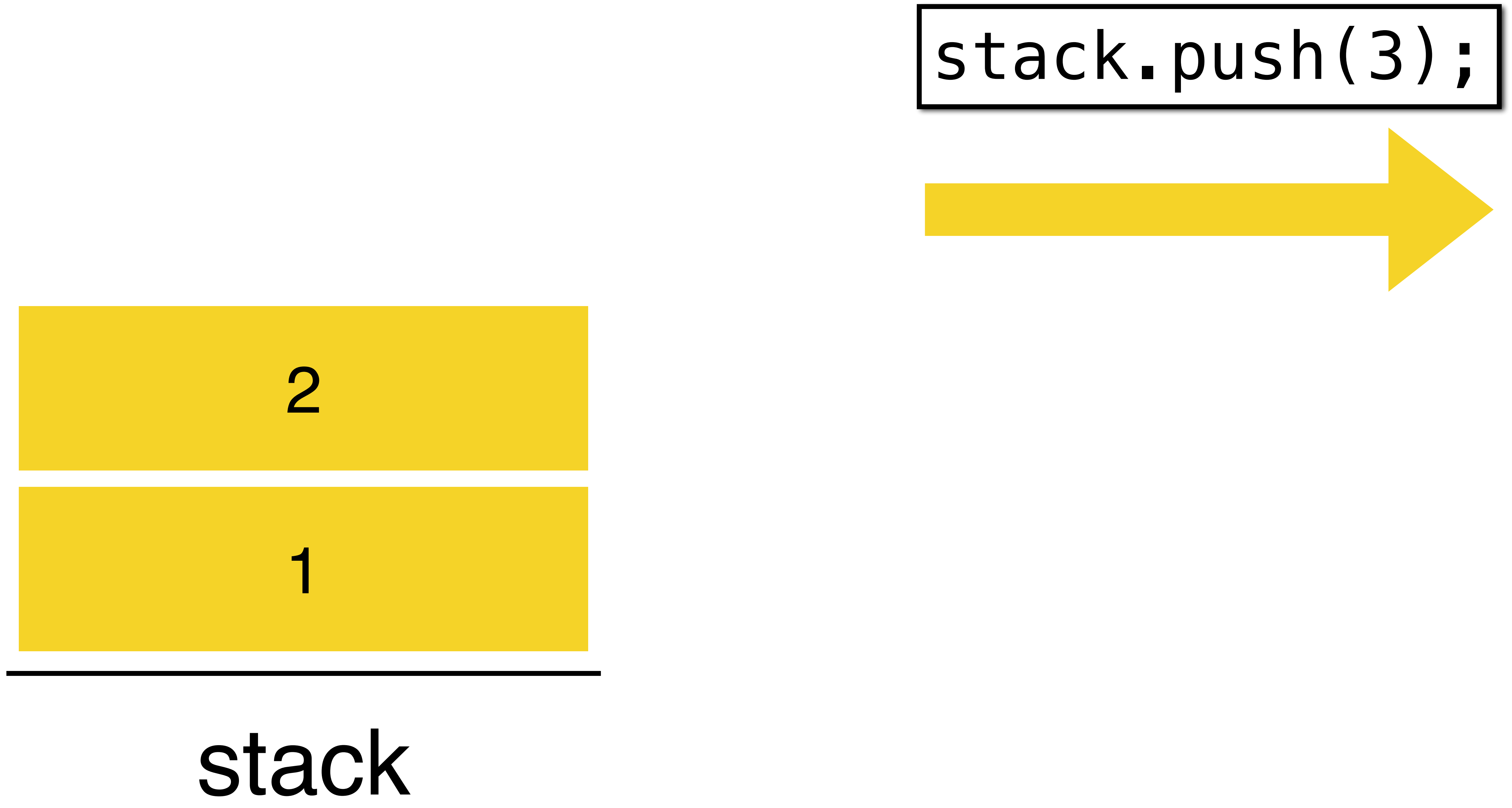
Stack visualization



```
stack.push(2);
```

Size: 1

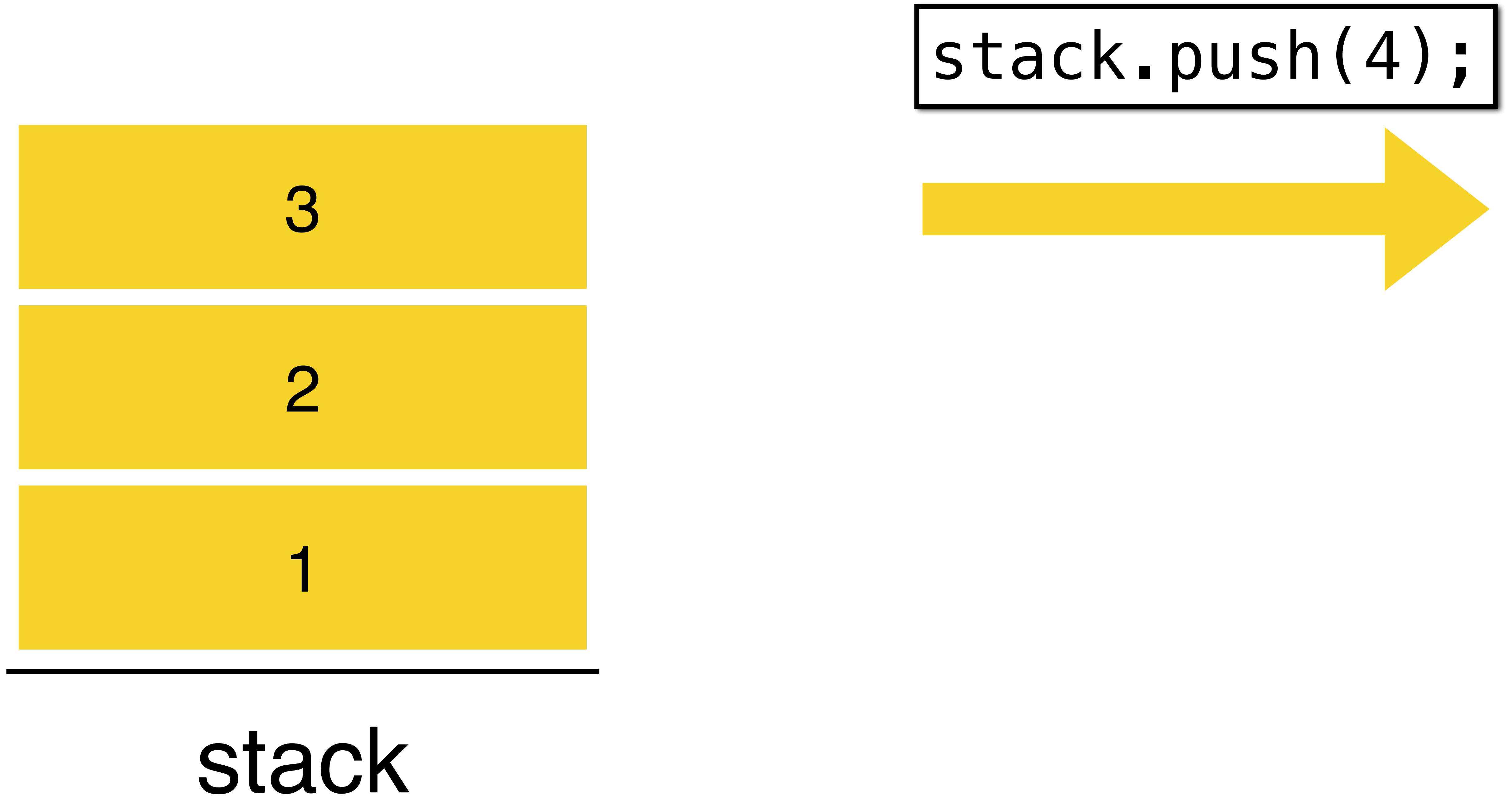
Stack visualization



```
stack.push(3);
```

Size: 2

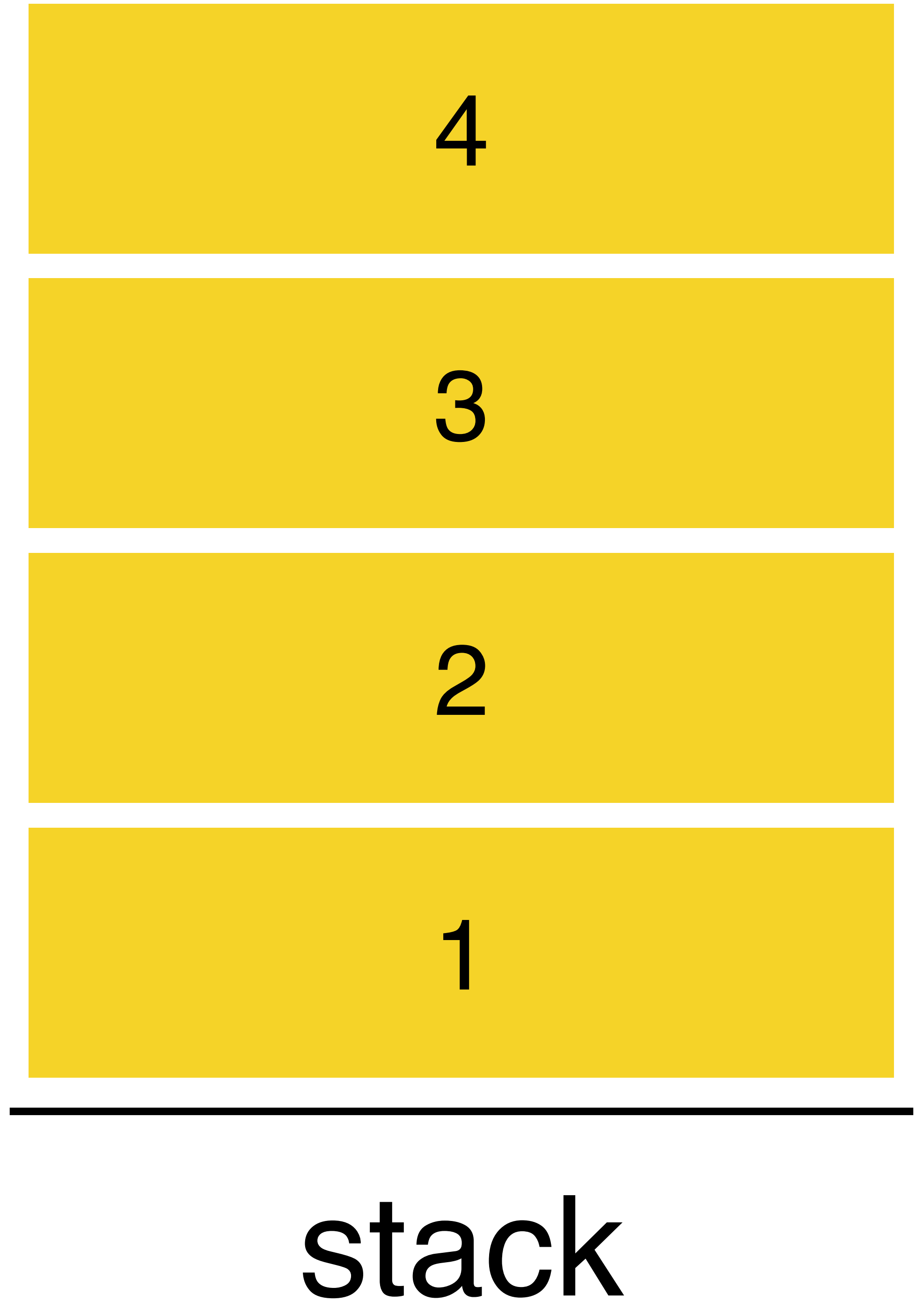
Stack visualization



```
stack.push(4);
```

Size: 3

Stack visualization


4

3

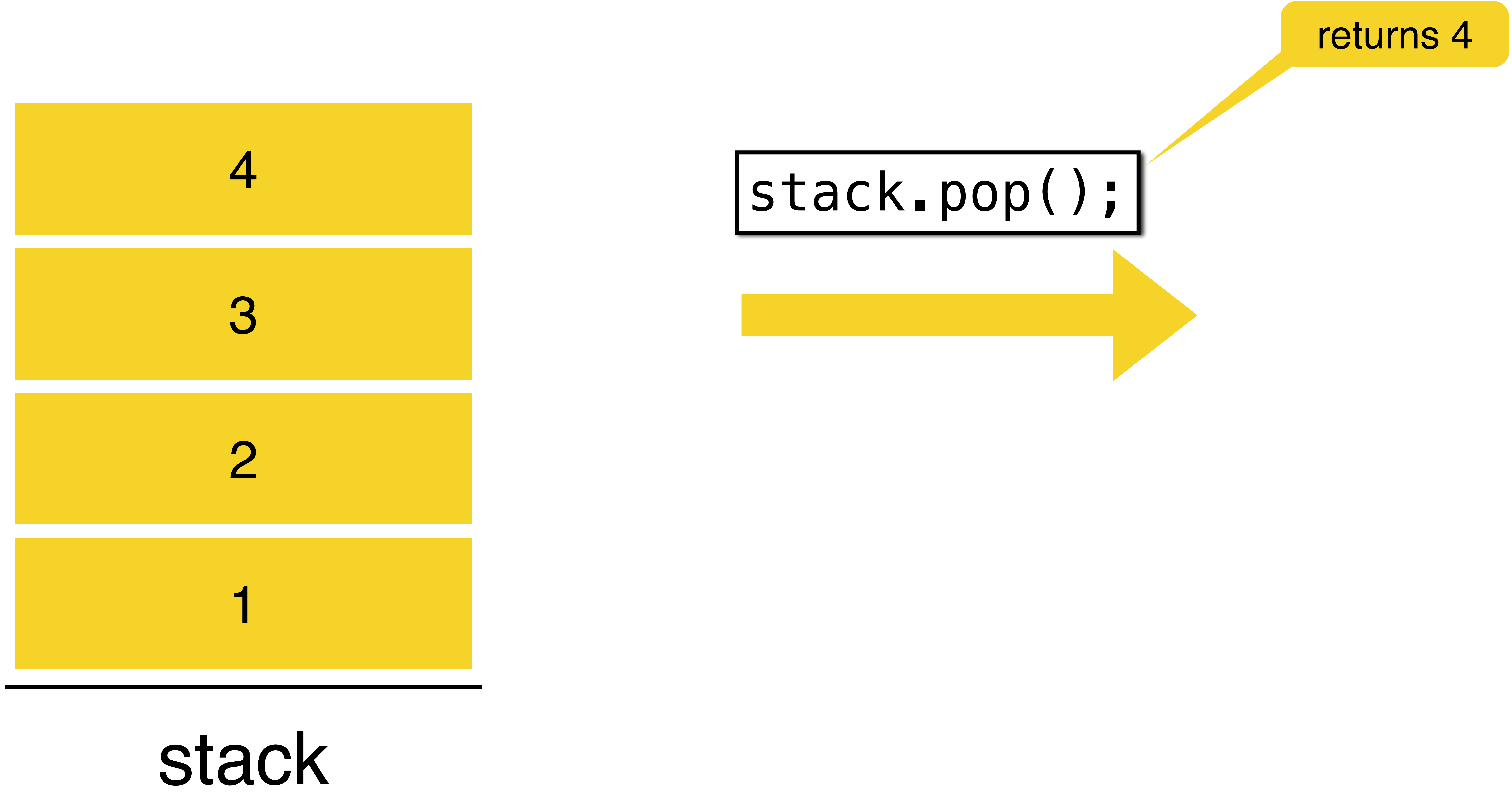
2

1

stack

Size: 4

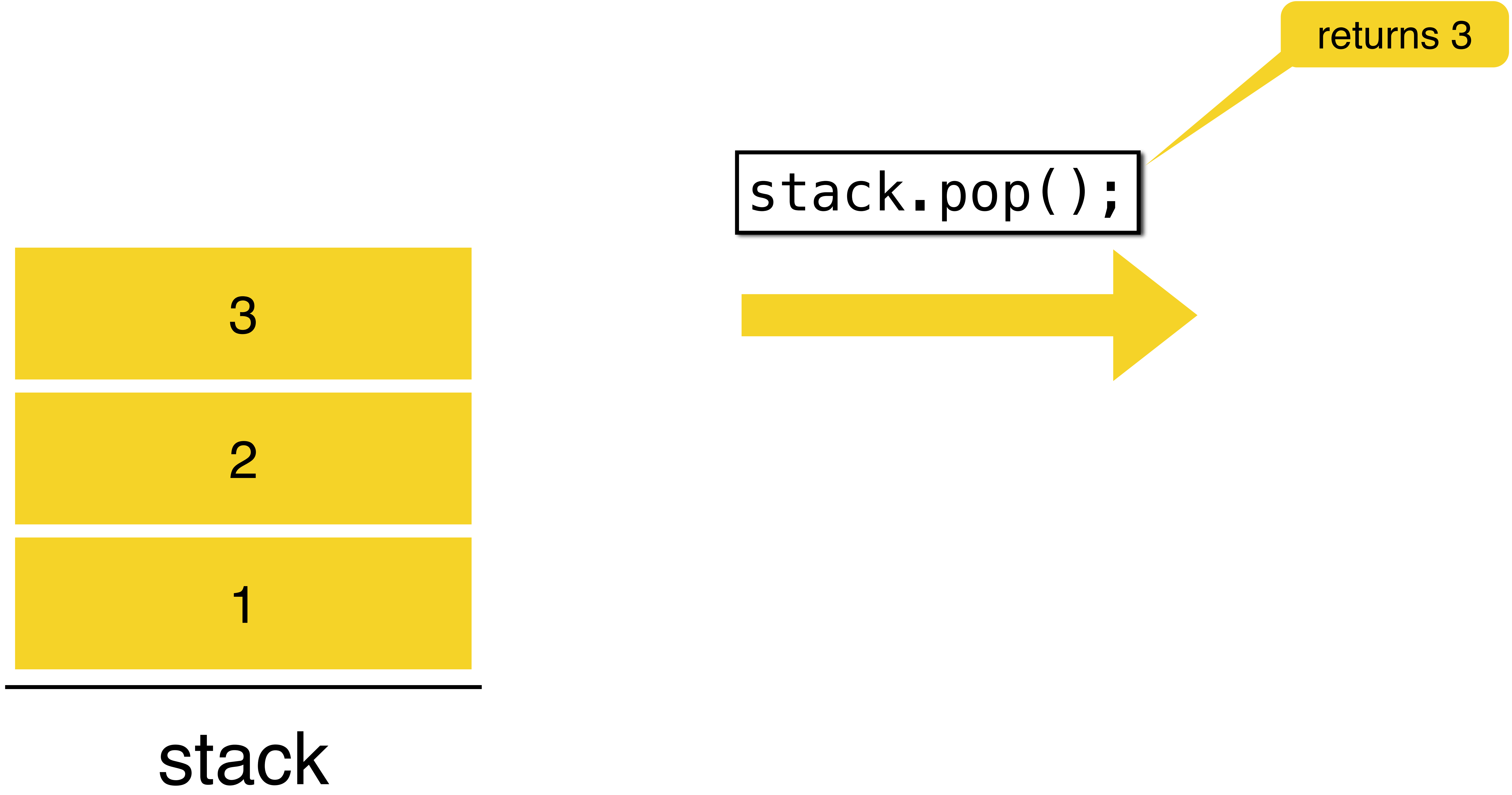
Stack visualization



```
stack.pop();
```

Size: 4

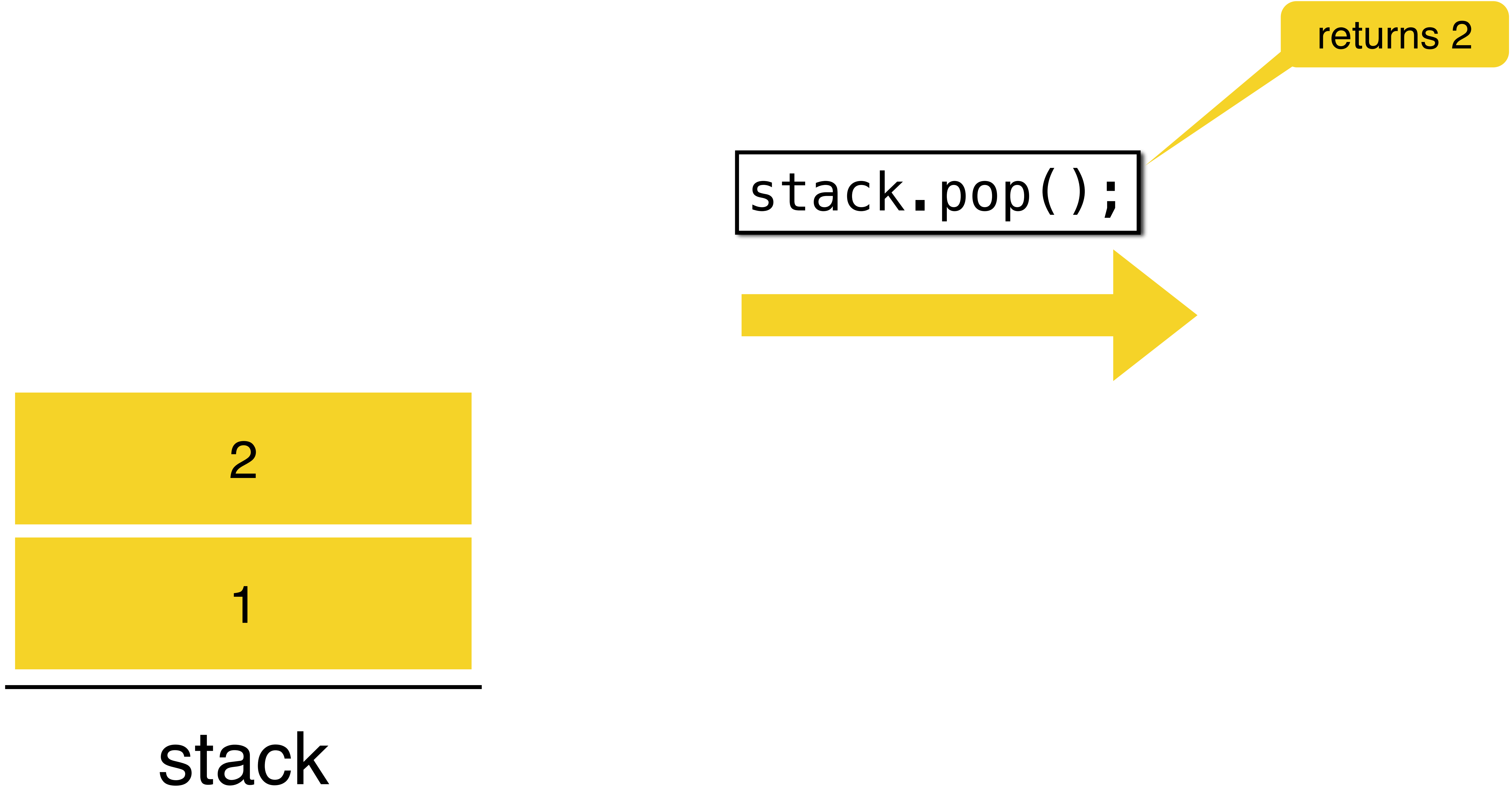
Stack visualization



```
stack.pop();
```

Size: 3

Stack visualization

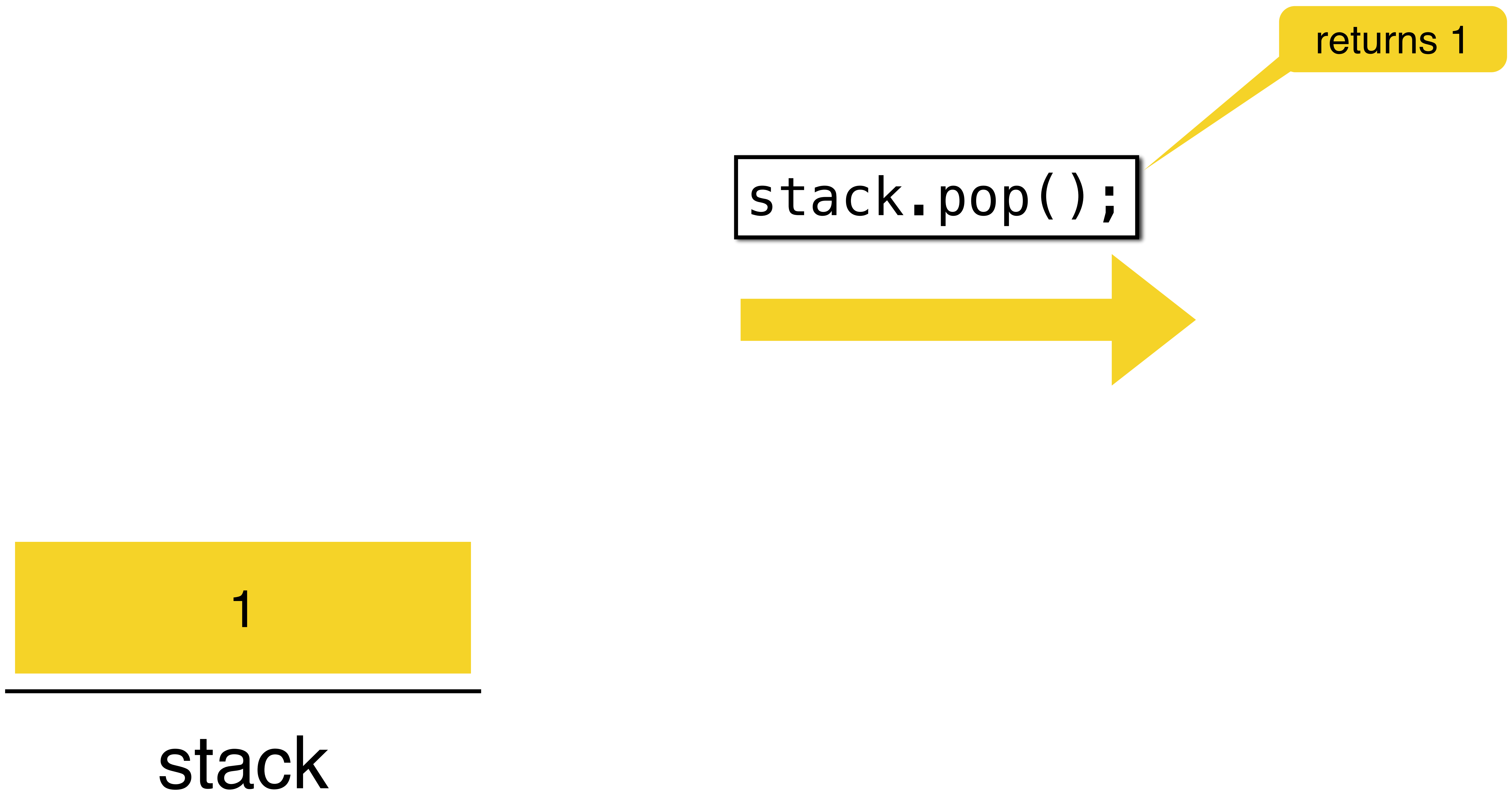


```
stack.pop();
```

returns 2

Size: 2

Stack visualization



returns 1

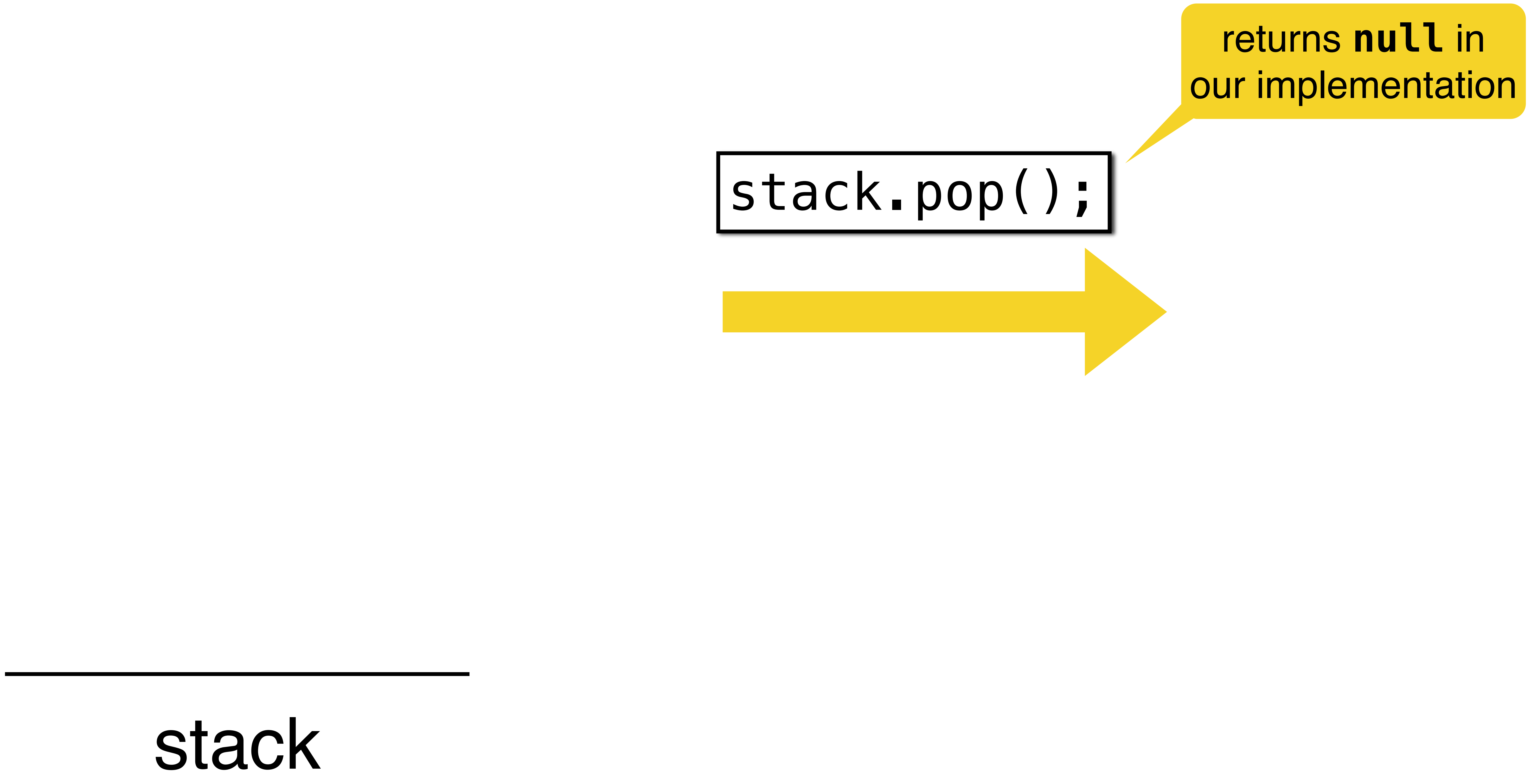
```
stack.pop();
```

1

stack

Size: 1

Stack visualization



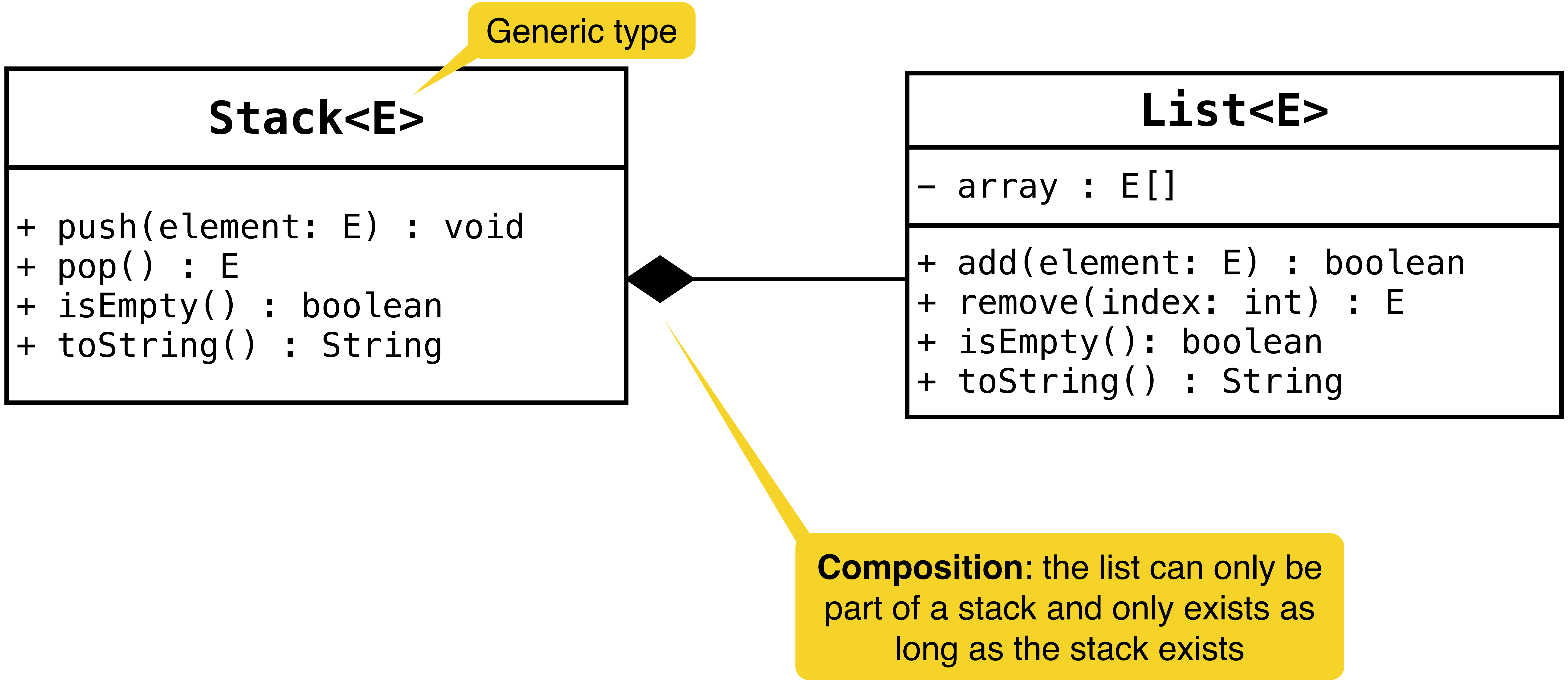
```
stack.pop();
```

returns **`null`** in our implementation

stack

Size: 0

Modeling a stack with a list



Implementation: stack with a list

```java
import org.checkerframework.checker.nullness.qual.*;
import java.util.*;

public class Stack<E> {

    private final List<E> list = new ArrayList<>();

    public void push(@NonNull E element) {
        list.add(element);
    }

    @Nullable
    public E pop() {
        if (isEmpty()) {
            return null;
        }
        return list.remove(list.size() - 1);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public String toString() {
        return list.toString();
    }
}
```

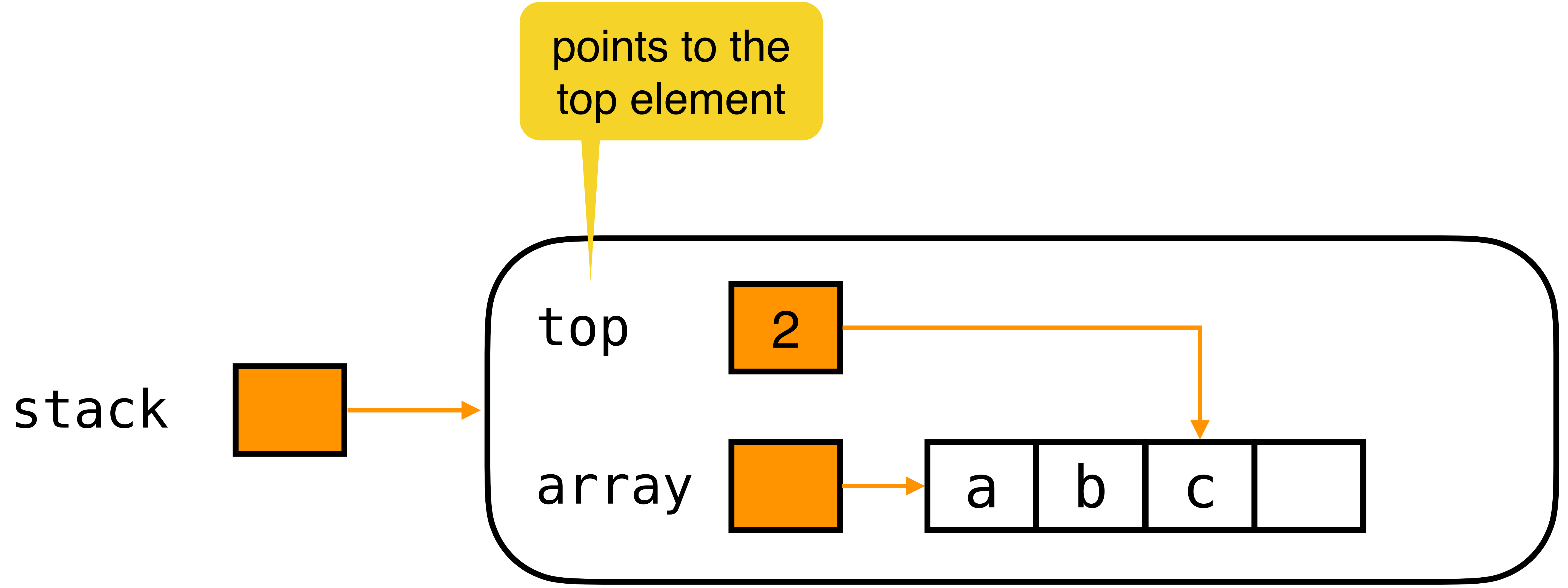
Add to the end of the list

Remove the last element

- Simple implementation
- Does not use all features of **`List`**

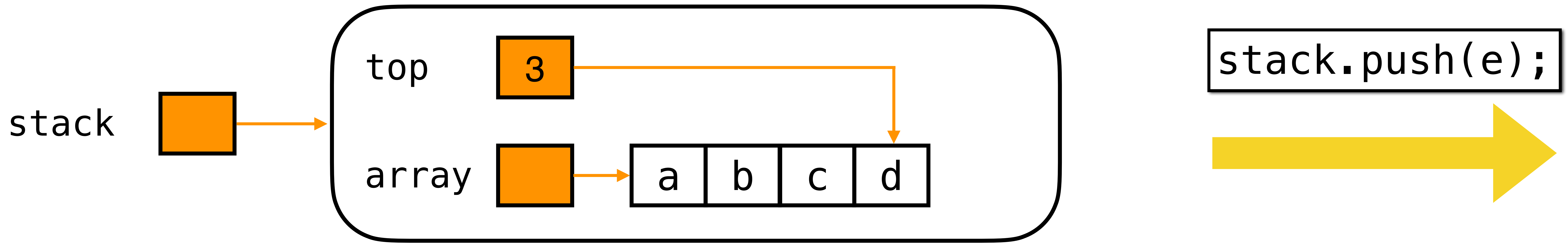
- Second idea
  - Realize the stack directly with an array and a pointer to the top occupied cell
  - If the array overflows, we replace it with a larger one

Example process: stack with an array



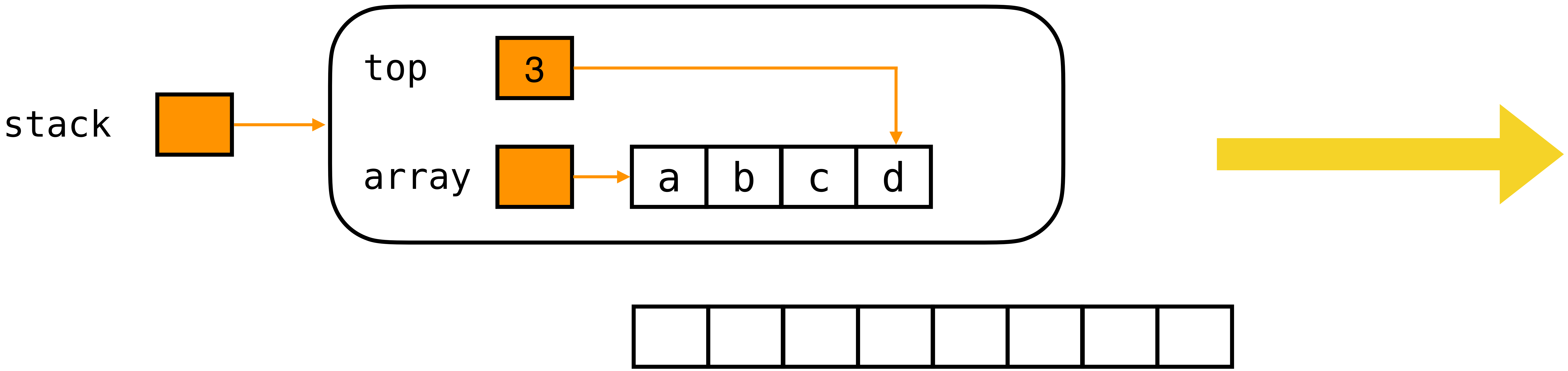
```
stack.push(d);
```

Example process: stack with an array

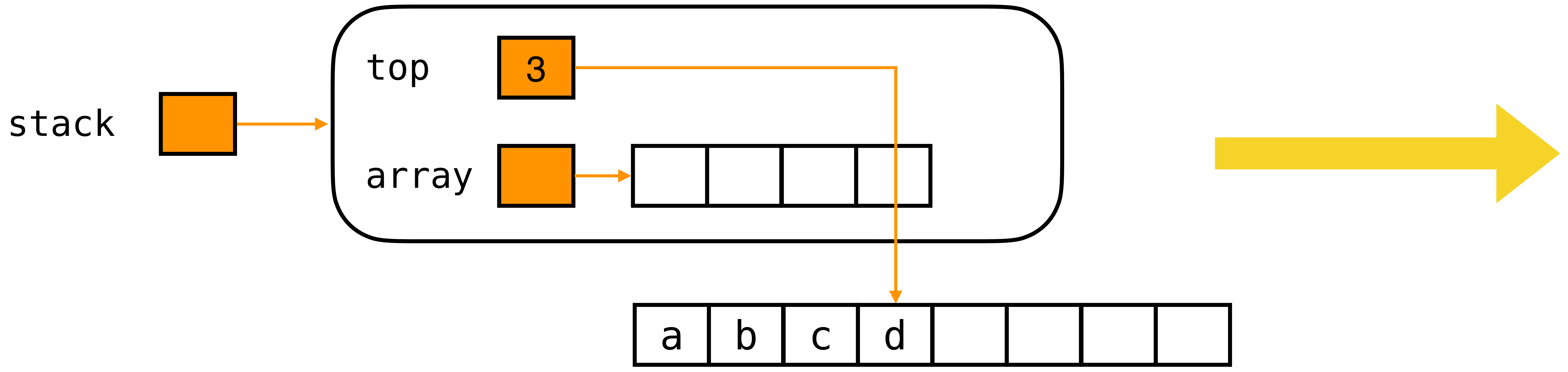


```
stack.push(e);
```

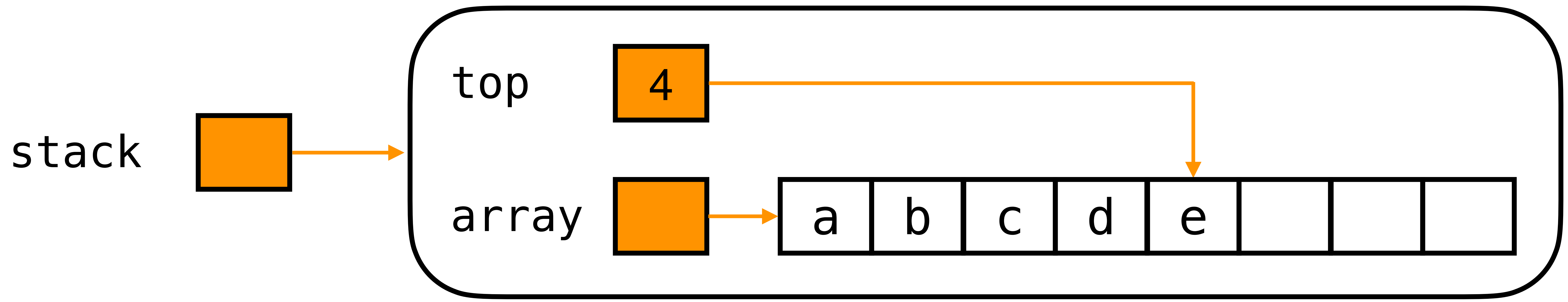
Example process: stack with an array



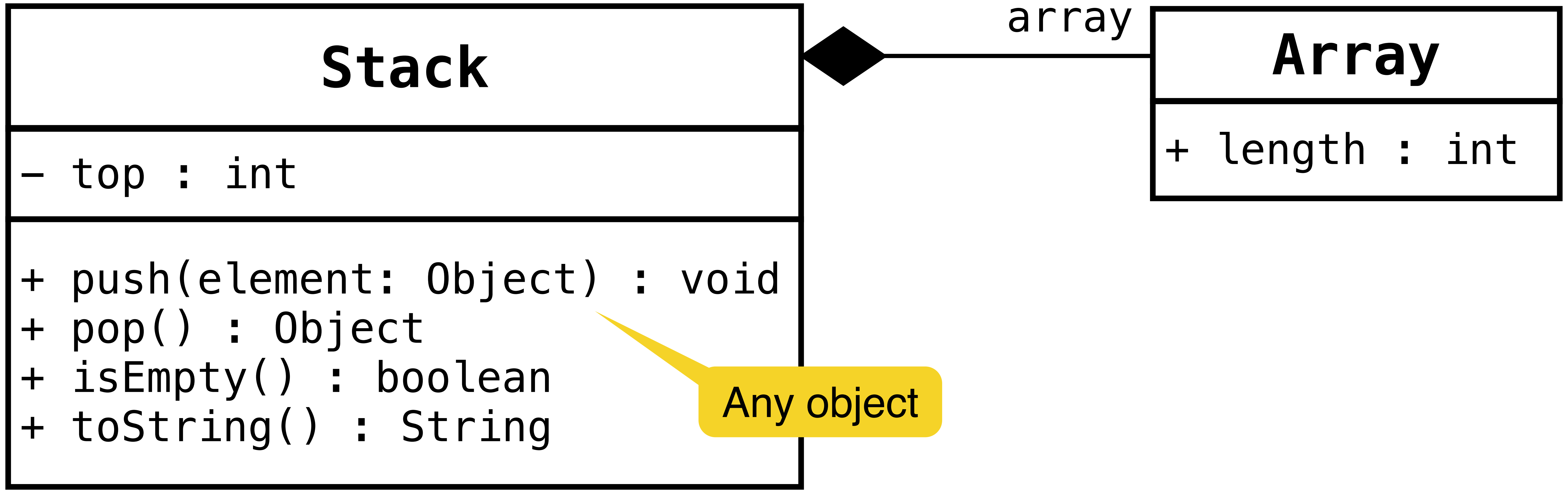
Example process: stack with an array



Example process: stack with an array



Modeling a stack with an array



Any object

Implementation: stack with an array

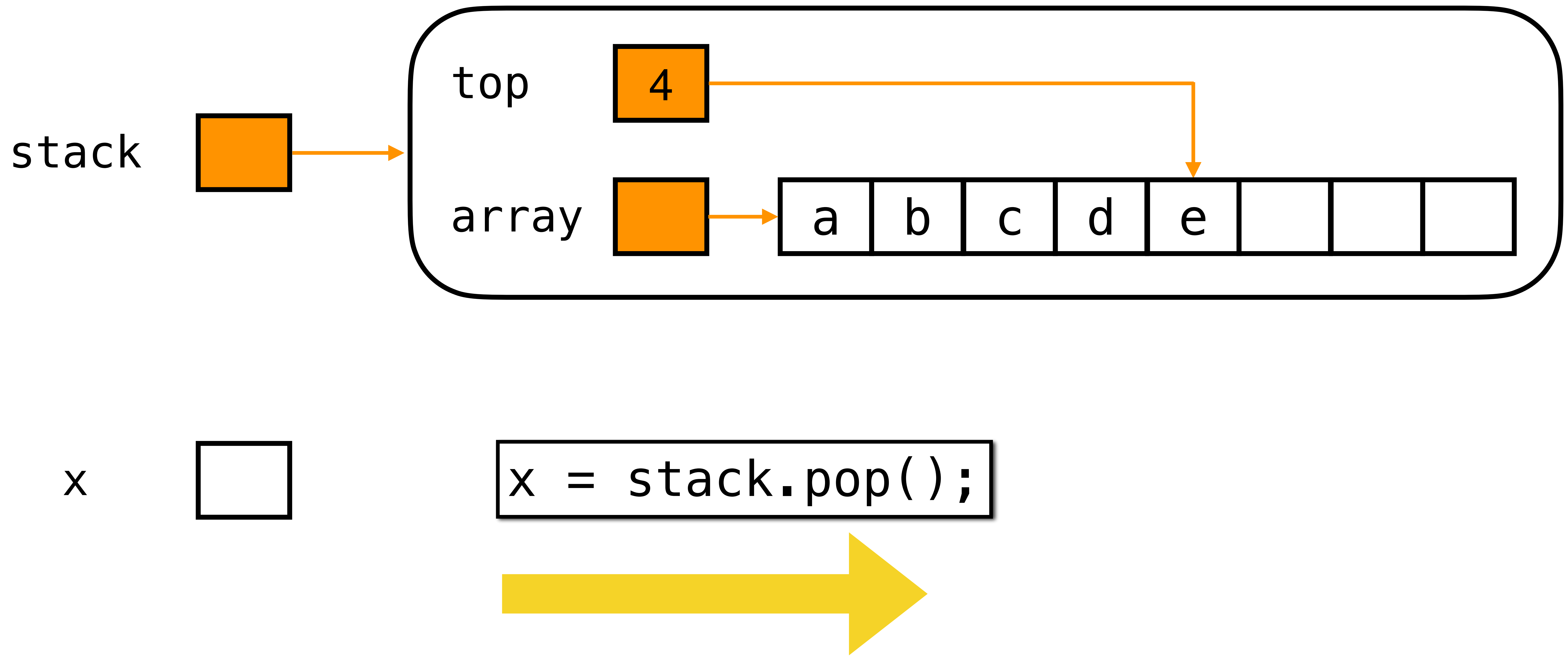
```java
import java.util.*;

public class Stack {
    private int top;
    private Object[] array;

    public Stack() {
        top = -1;
        array = new Object[4];
    }
    public boolean isEmpty() {
        return top < 0;
    }
    public void push(Object element) {
        top++;
        if (top == array.length) {
            Object[] newArray = new Object[2 * top];
            for(int i = 0; i < top; i++) {
                newArray[i] = array[i];
            }
            array = newArray;
        }
        array[top] = element;
    }
    public Object pop() {     // Assumption top > -1
        return array[top--];
    }
    public String toString() {
        return Arrays.toString(array);
    }
}
```

- Disadvantage: new space is allocated but never released
- The implementation is not type safe

- **Idea:** if the length drops to half again, we release it

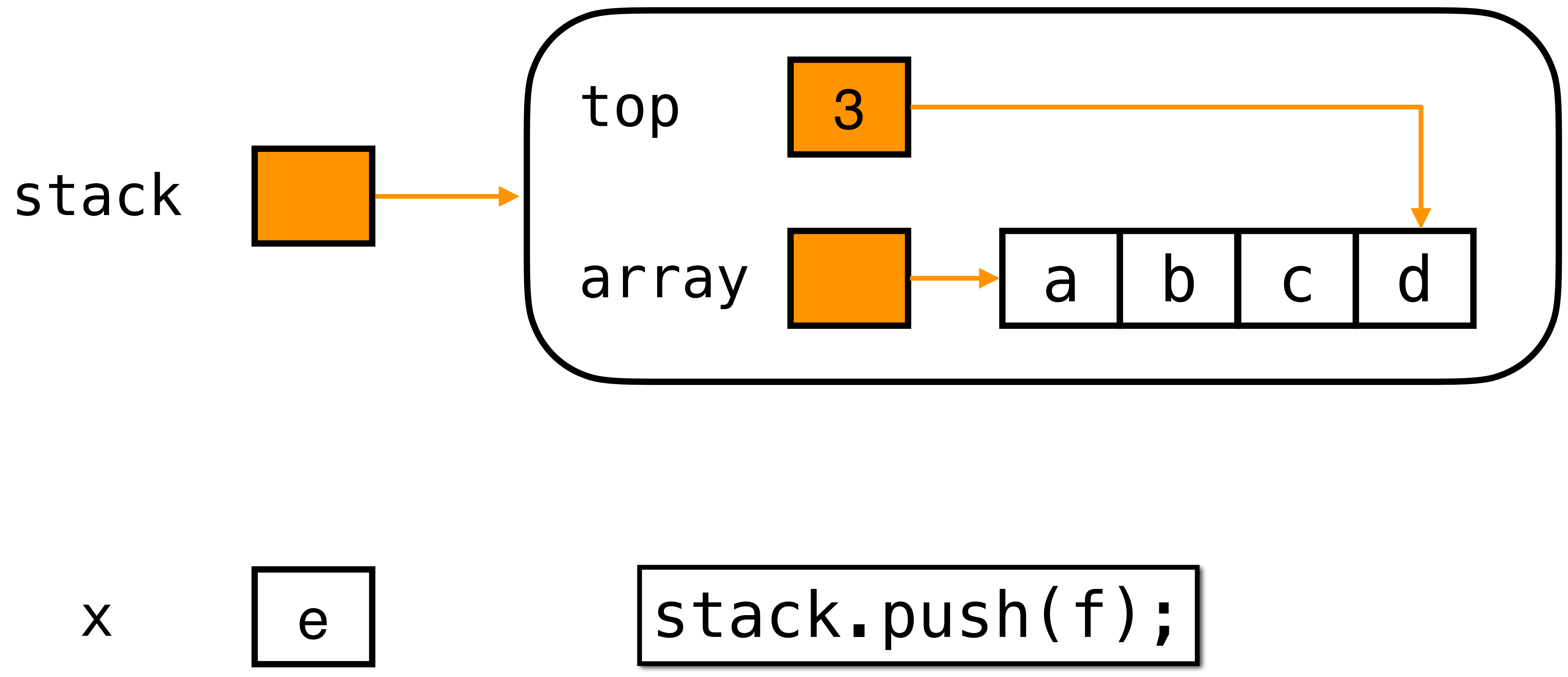
Double the array size, copy all existing elements into a new array and use this one from now on

Example process: stack with an array



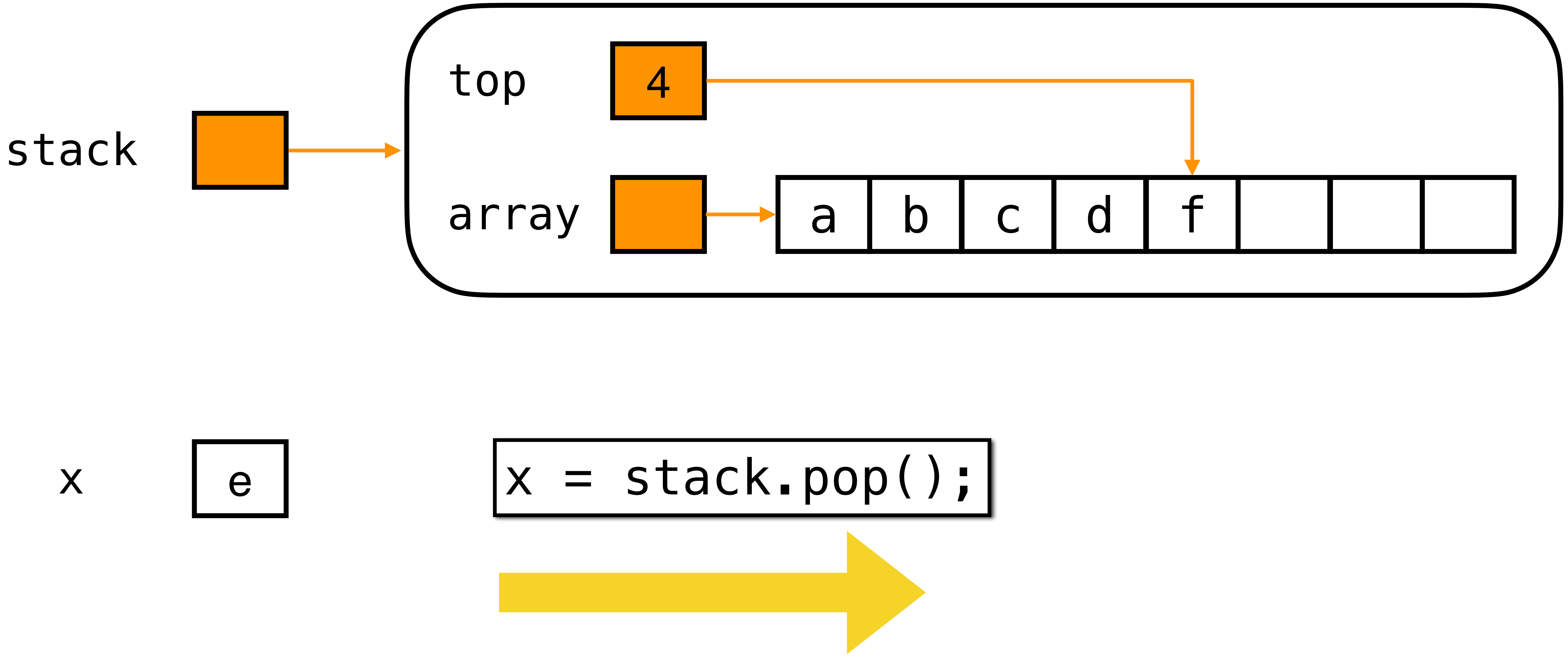
```
x = stack.pop();
```

Example process: stack with an array



```
stack.push(f);
```

Example process: stack with an array



stack

top 4

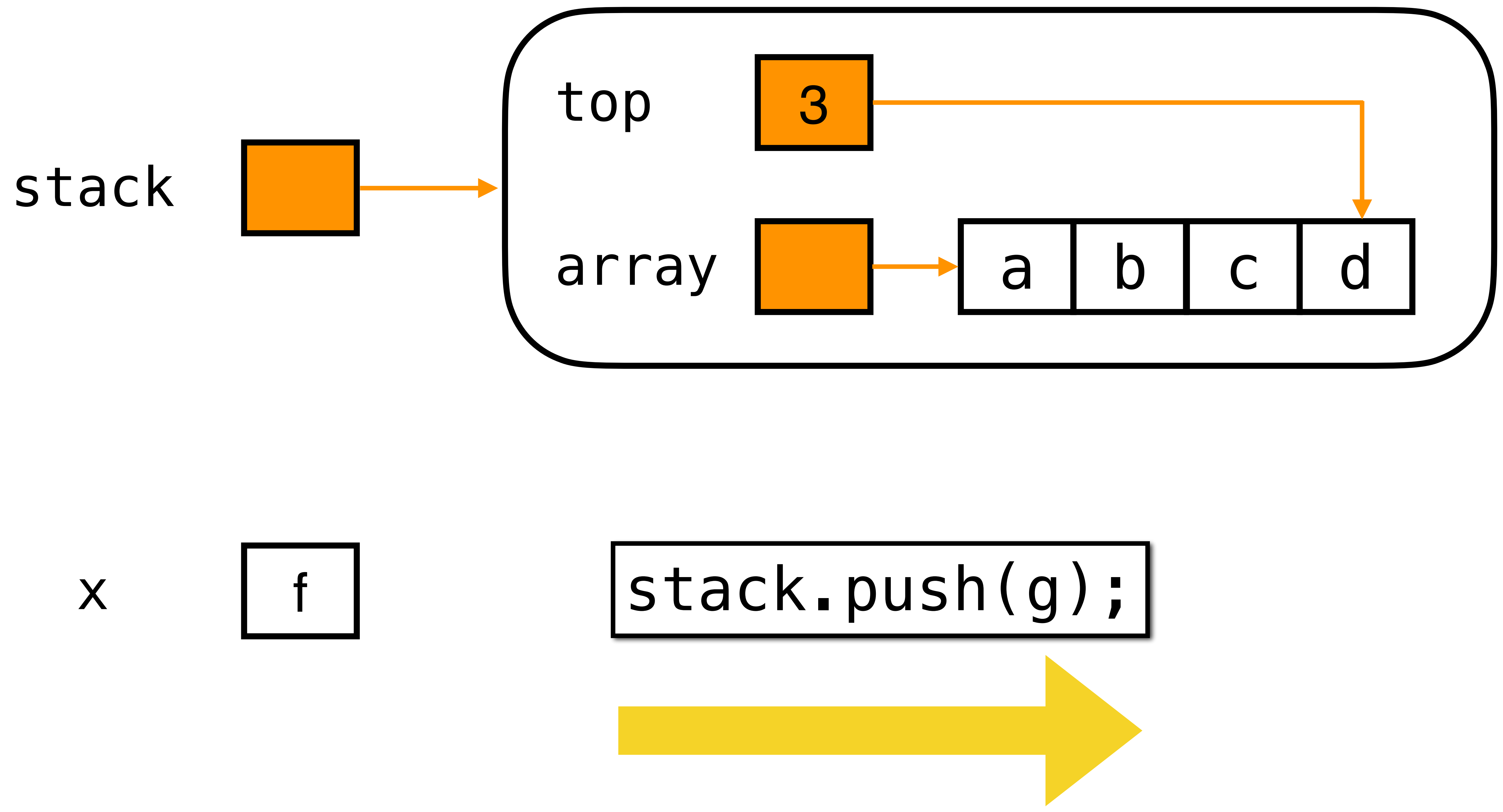
array

a b c d f

x e

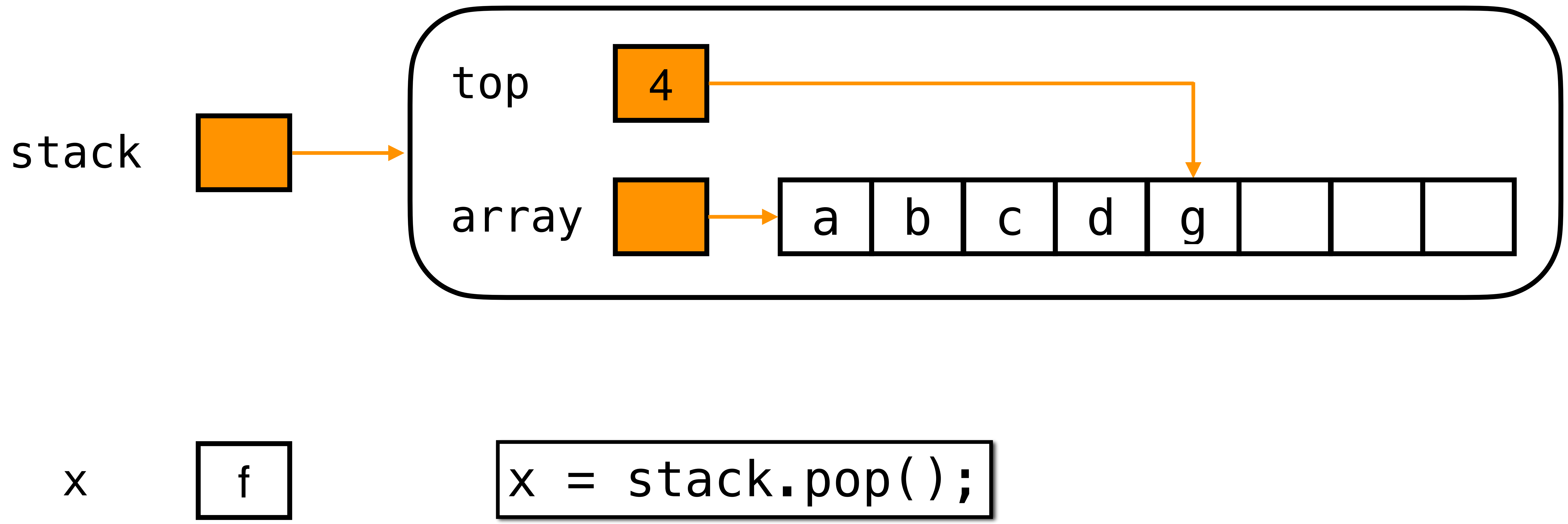
```
x = stack.pop();
```

Example process: stack with an array



```
stack.push(g);
```

Example process: stack with an array

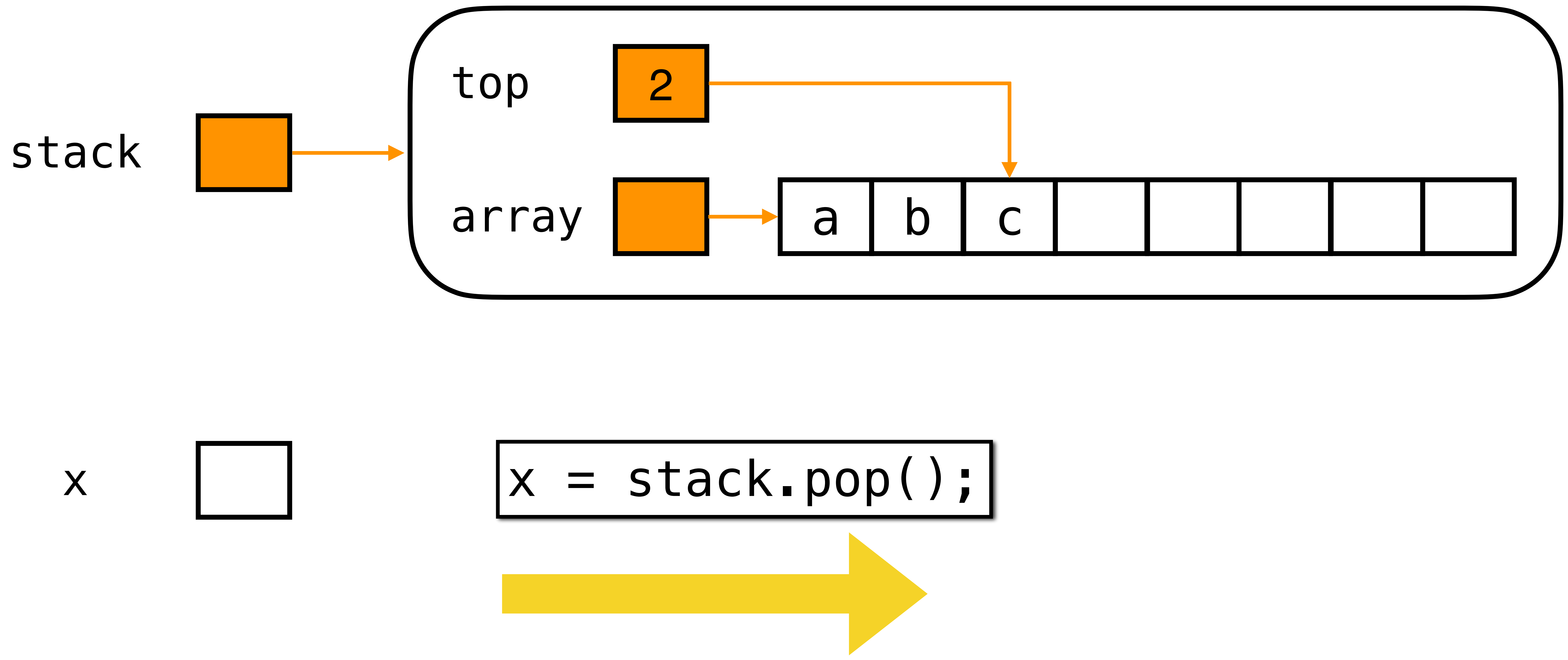


```
x = stack.pop();
```

Observation

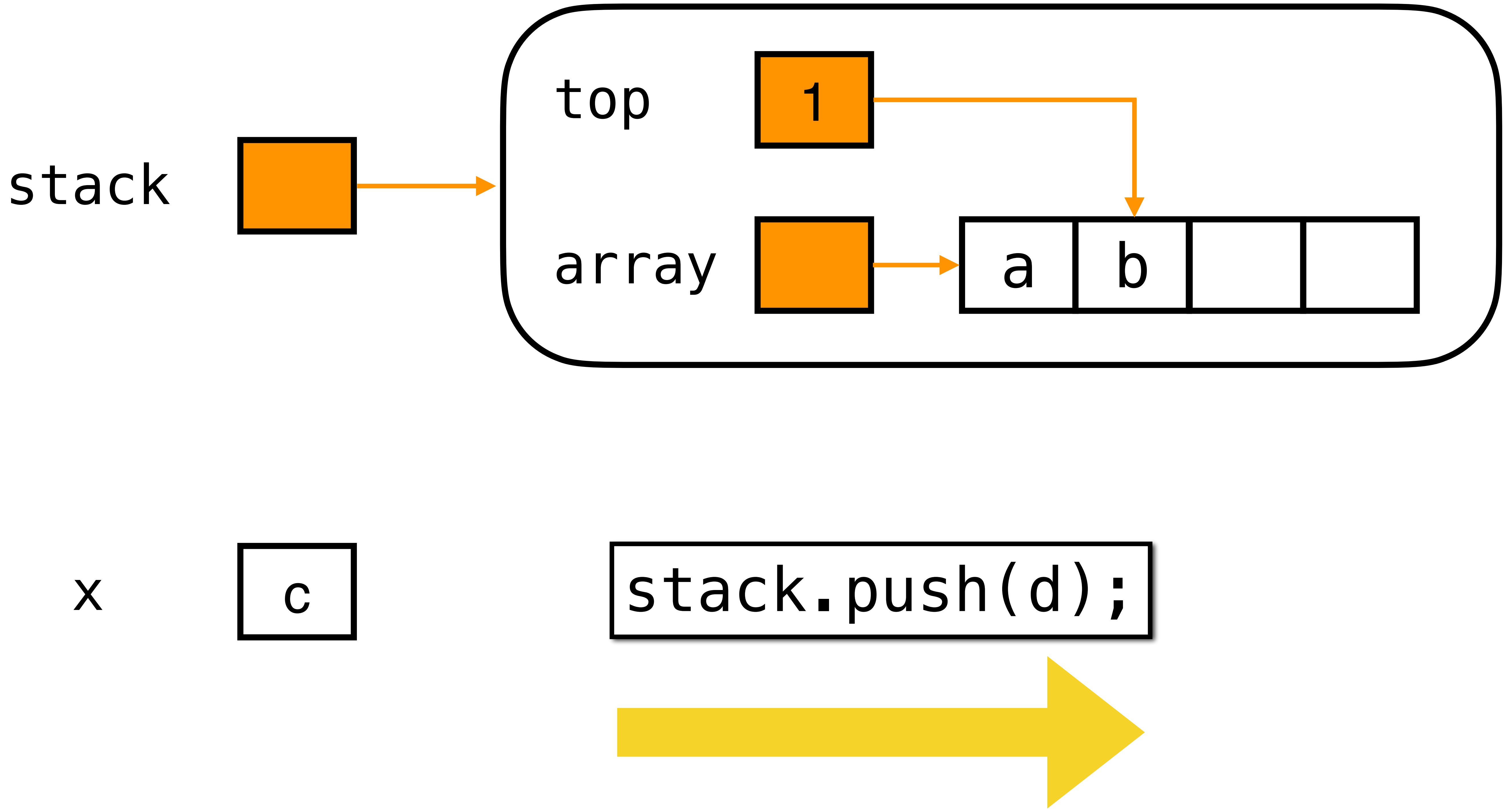
- In the worst case, all elements must be copied for each operation
- Idea: the stack only releases when the length drops to a quarter - and then only half

Example process: stack with an array



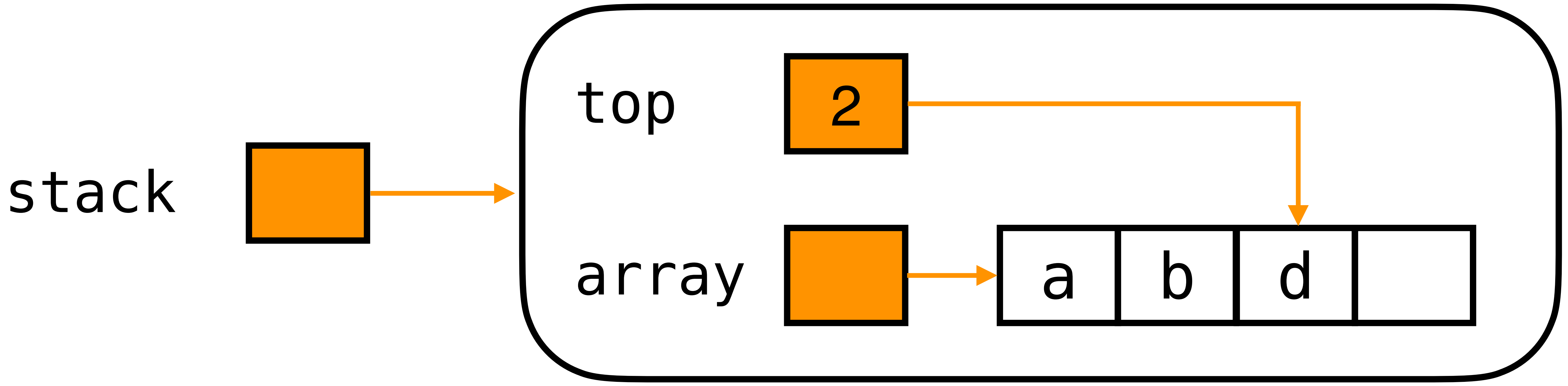
```
x = stack.pop();
```

Example process: stack with an array



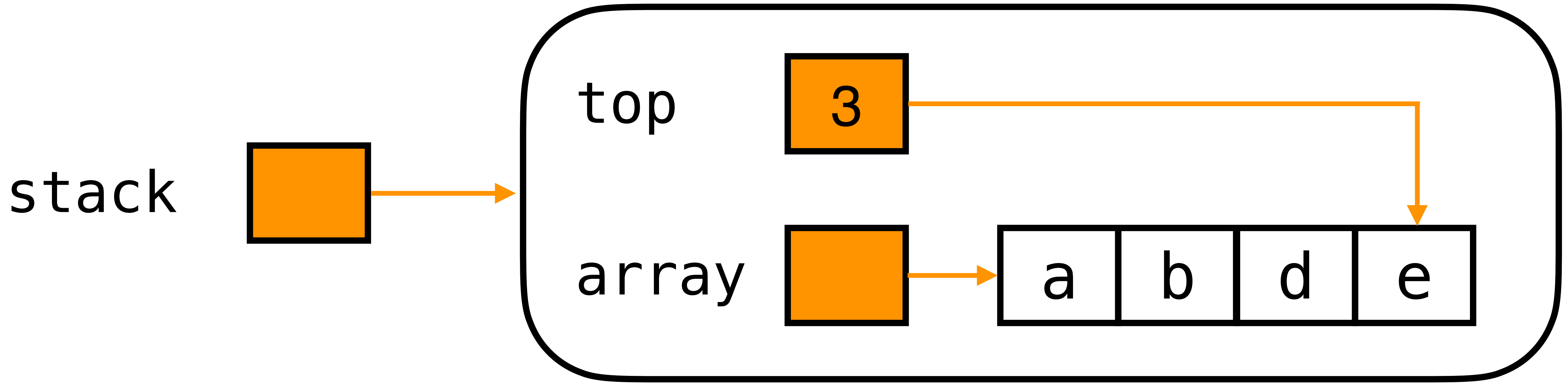
```
stack.push(d);
```

Example process: stack with an array



```
stack.push(e);
```

Example process: stack with an array

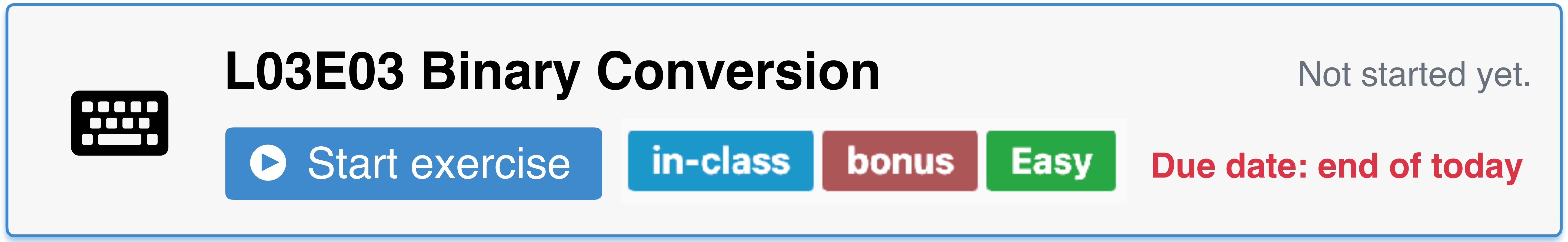


Analysis

- Before each copy, at least half as many operations are performed as elements are copied
- Averaged over the entire sequence of operations, a maximum of two numbers are copied per operation (amortized effort analysis)

```
public Object pop() {
    // Assumption top > -1
    Object result = array[top];
    if (top == array.length / 4 && top >= 2) {
        Object[] newArray = new Object[2 * top];
        for(int i = 0; i < top; i++) {
            newArray[i] = array[i];
        }
        array = newArray;
    }
    top--;
    return result;
}
```

Covered in the course
Fundamentals of Algorithms and Data Structures


L03E03 Binary Conversion

Not started yet.

Due date: end of today

10 min

3 pts

- **Problem**: Convert a positive integer number into its binary representation
  - Read the integer from the console
  - Print the binary representation to the console
  - You have to use the **`Stack`** class (based on **`java.util.List`**)
  - Example input 8 would lead to 1000, input 9 would lead to 1001, etc.

```java
public static void convertNumberToBinary() {
    int number = InputReader.readInt("Enter the number: ");
    Stack<Integer> stack = new Stack<>();

    // TODO: implement

}
```

Example solution

```java
public static void convertNumberToBinary() {
    int number = InputReader.readInt("Enter the number: ");
    Stack<Integer> stack = new Stack<>();

    while (number > 0) {
        stack.push(number % 2);
        number = number / 2;
    }

    while (!stack.isEmpty()) {
        System.out.print(stack.pop());
    }

}
```

1. **while loop**: push what remains (0 or 1) to the stack

2. **while loop**: pop all elements until the stack is empty

Break

10 min

The lecture will continue at **16:00**

Outline

- List
- Stack
- ➡ **Queue**

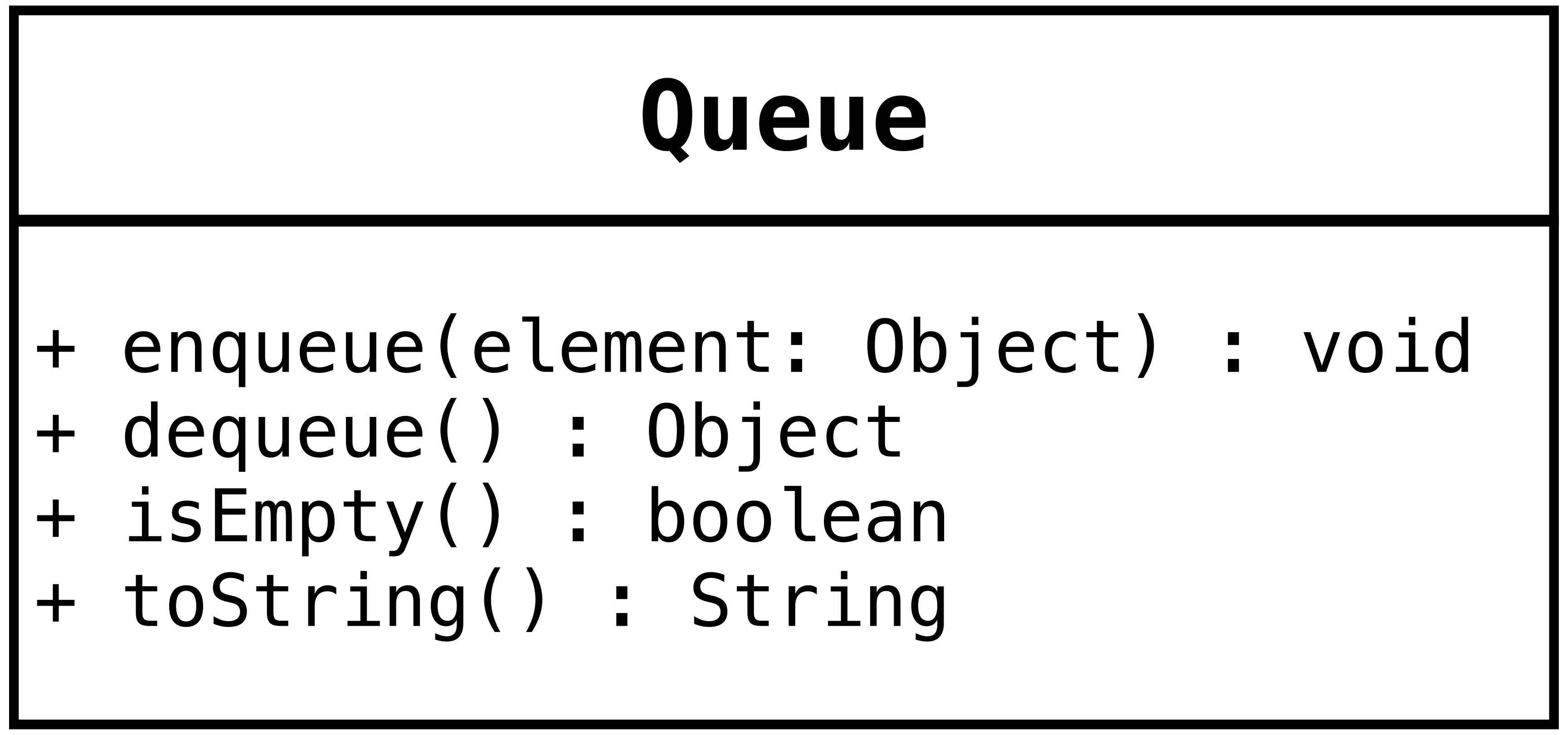
Queue operations

- (Waiting) queues manage their elements according to the **FIFO** principle: First In First Out
  - Stacks on the other hand use the **LIFO** principle: Last In First Out
- Operations

`void enqueue(Object element)`	adds the **`element`** to the queue
`Object dequeue()`	returns the **first** element
`boolean isEmpty()`	tests for emptiness
`String toString()`	returns a string representation

- Ability to create an empty queue

Modeling a queue



Queue

```
+ enqueue(element: Object) : void
+ dequeue() : Object
+ isEmpty() : boolean
+ toString() : String
```

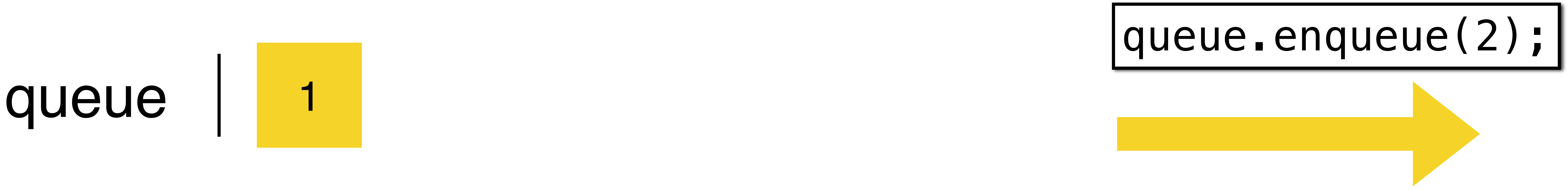
Queue visualization

queue

```
queue.enqueue(1);
```

Size: 0

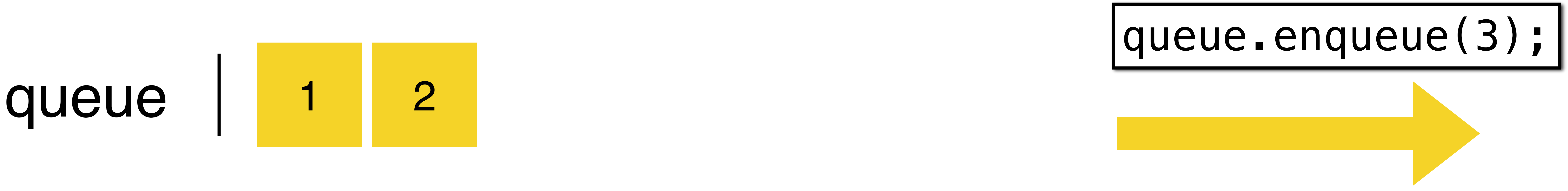
Queue visualization



```
queue.enqueue(2);
```

Size: 1

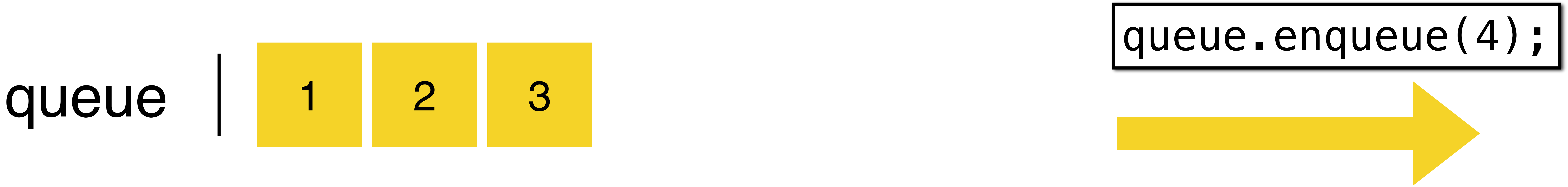
Queue visualization



```
queue.enqueue(3);
```

Size: 2

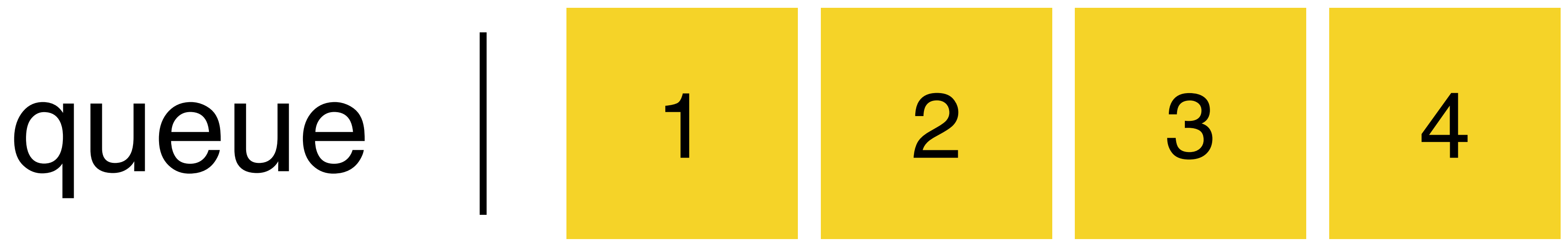
Queue visualization



```
queue.enqueue(4);
```

Size: 3

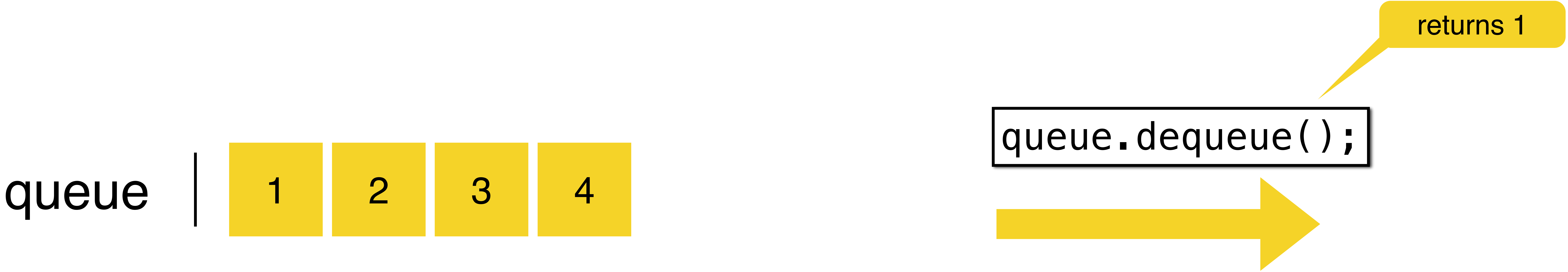
Queue visualization



queue | 1 2 3 4

Size: 4

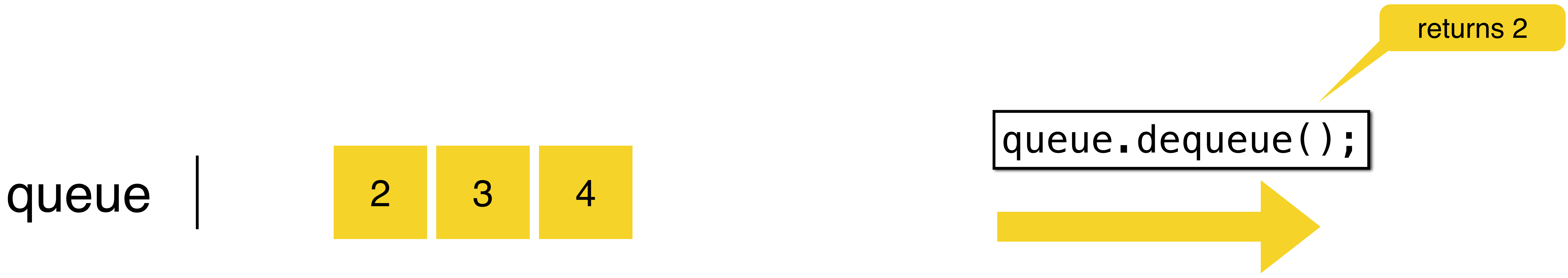
Queue visualization



```
queue.dequeue();
```

Size: 4

Queue visualization



queue

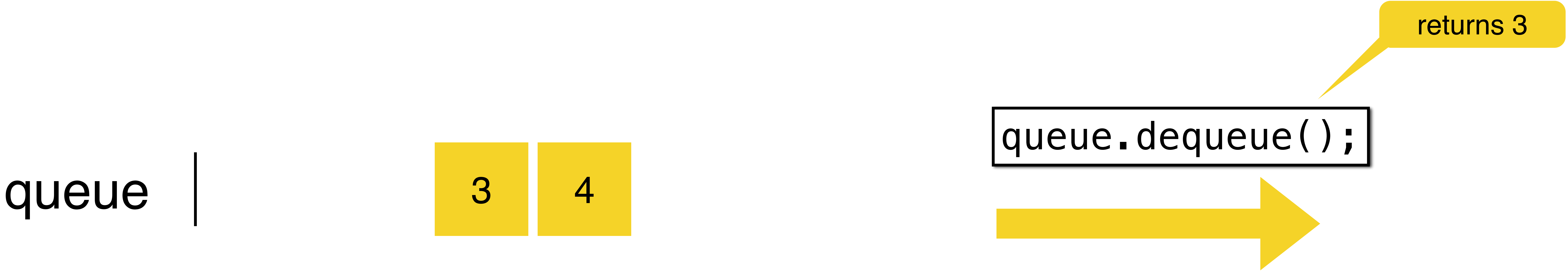
2	3	4

```
queue.dequeue();
```

returns 2

Size: 3

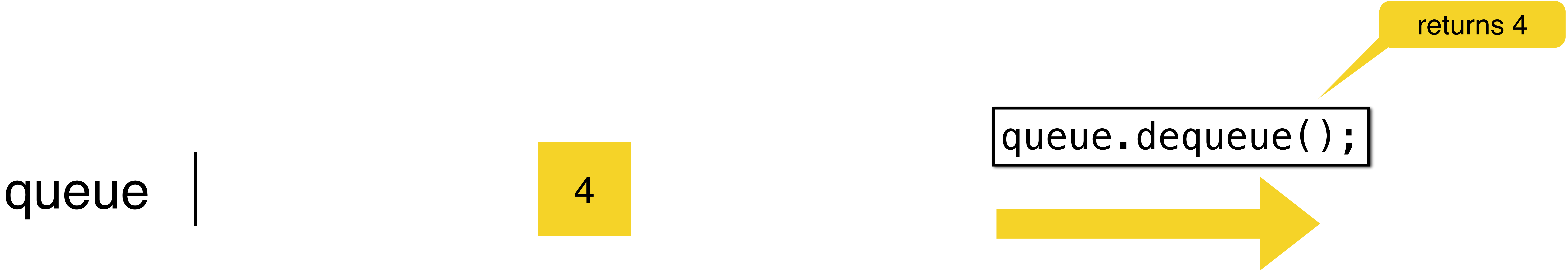
Queue visualization



```
queue.dequeue();
```

Size: 2

Queue visualization



```
queue.dequeue();
```

Size: 1

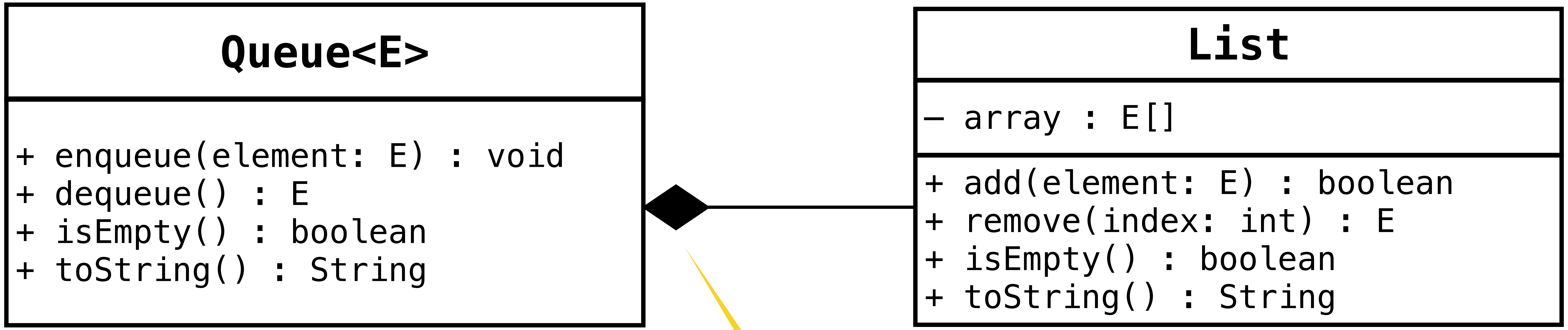
Queue visualization

```
queue.dequeue();
```

returns **`null`**

Size: 0

Modeling a queue with a **list**



Composition: the list can only be part of a queue and only exists as long as the queue exists

Implementation: queue with a `list`

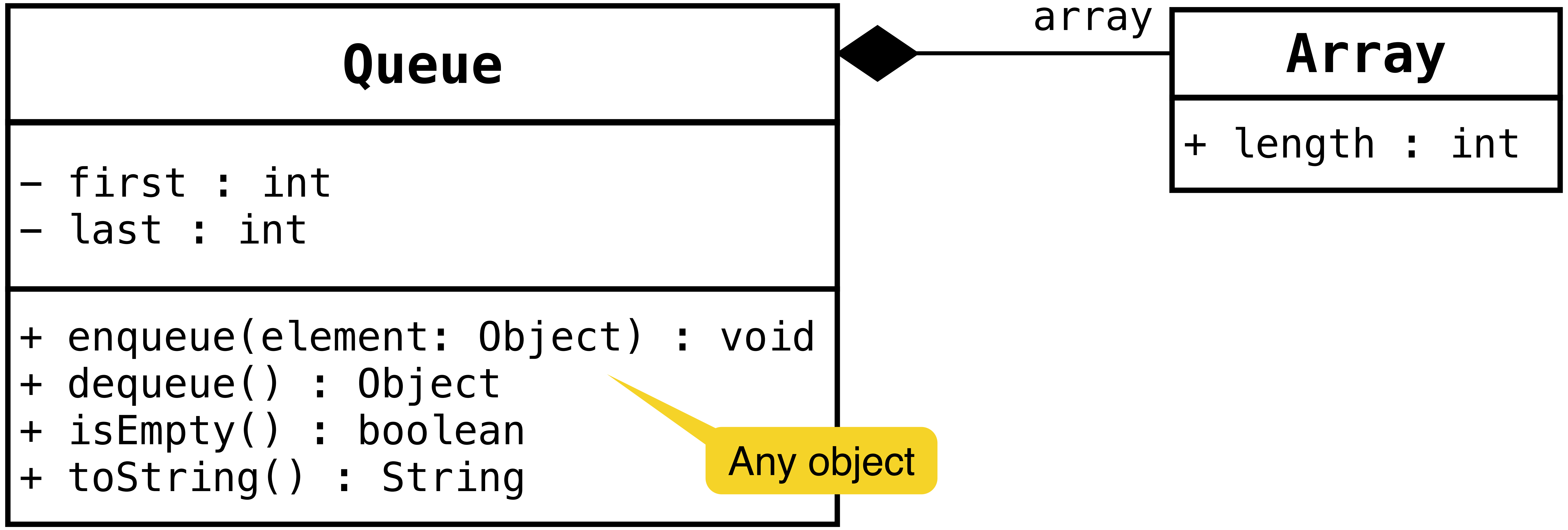
```java
import org.checkerframework.checker.nullness.qual.*;
import java.util.*;

public class Queue<E> {

    @NonNull
    private final List<E> list = new ArrayList<>();

    public void enqueue(@NonNull E item) {
        list.add(item);
    }

    @Nullable
    public E dequeue() {
        if (isEmpty()) {
            return null;
        }
        return list.remove(0);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public String toString() {
        return list.toString();
    }
}
```

Add to the end of the list

Remove the first element

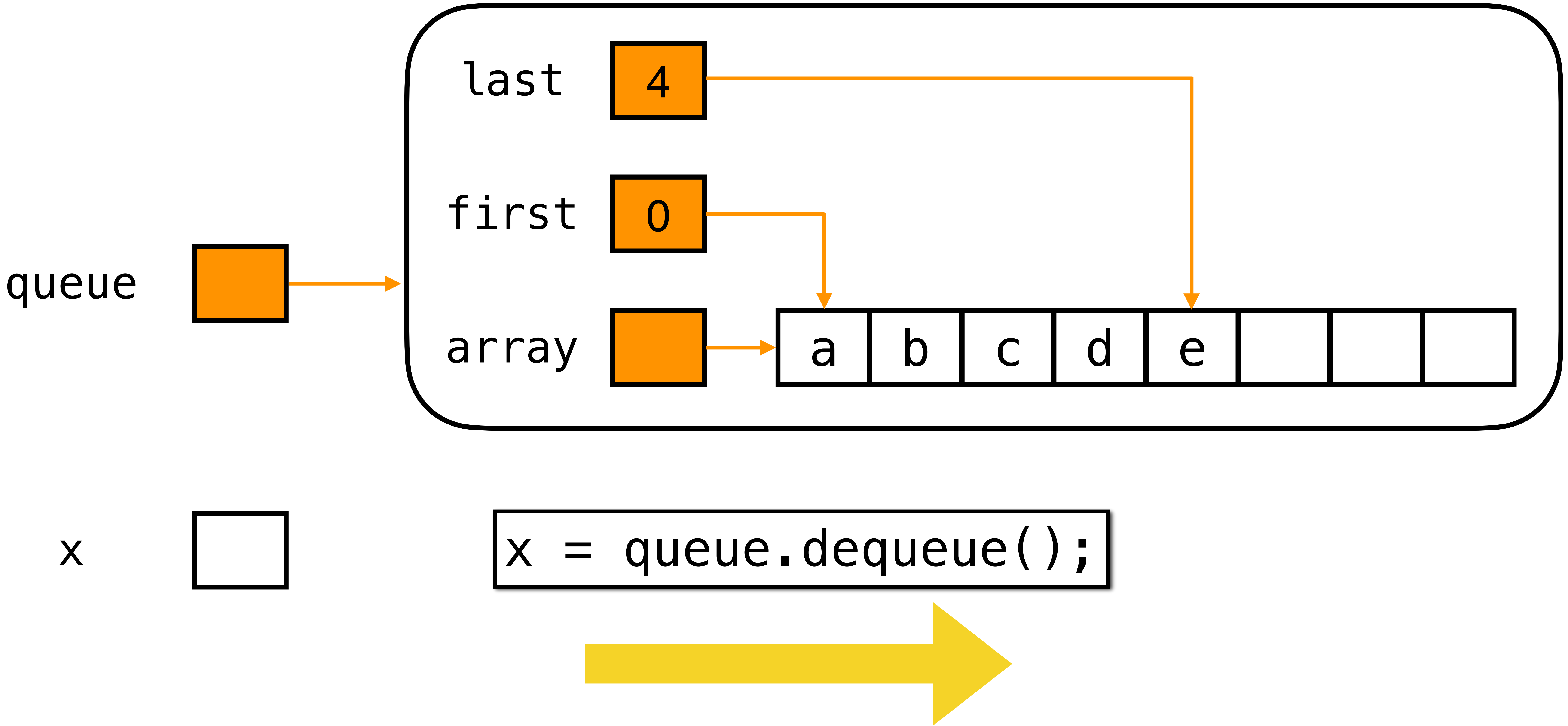
- Analysis
  - Simple implementation
  - Does not use all features of **`List`**
- Second idea
  - Realize the queue directly using an array
  - If the array overflows, we replace it with a larger one

Modeling a queue with an `array`



Any object

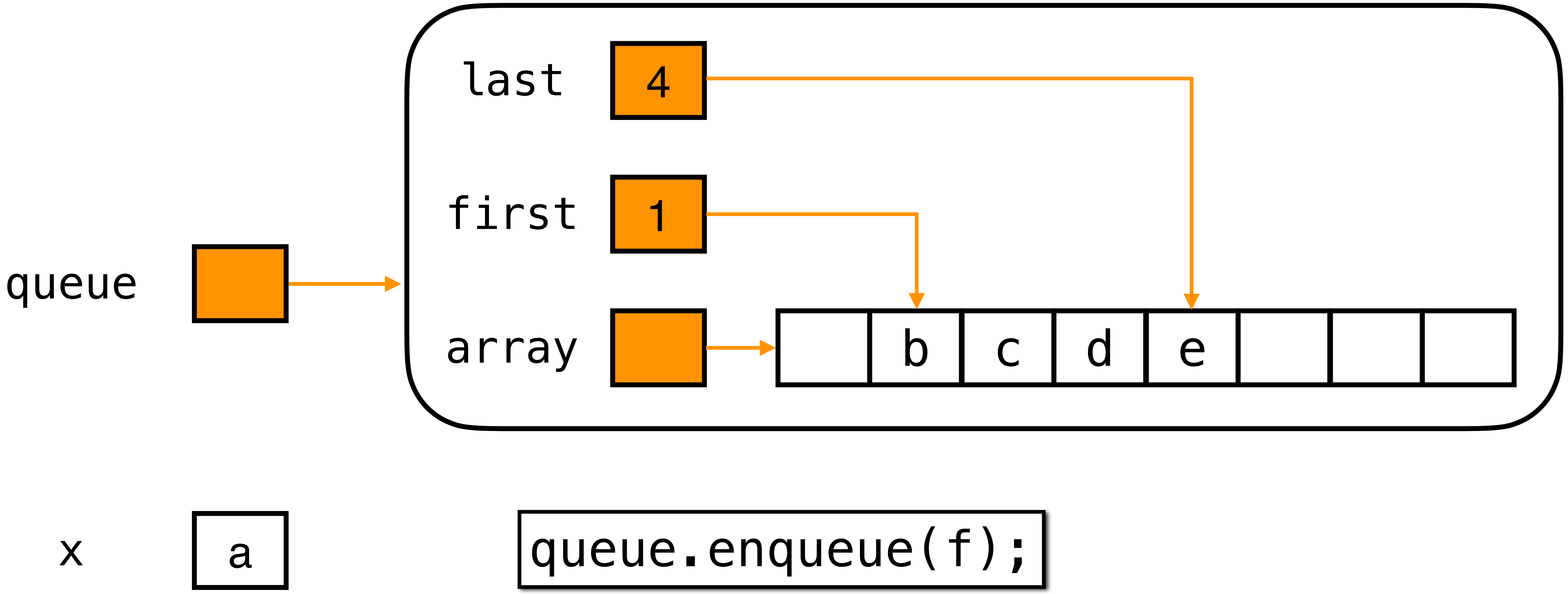
Example process: queue with an **`array`**



```
x = queue.dequeue();
```

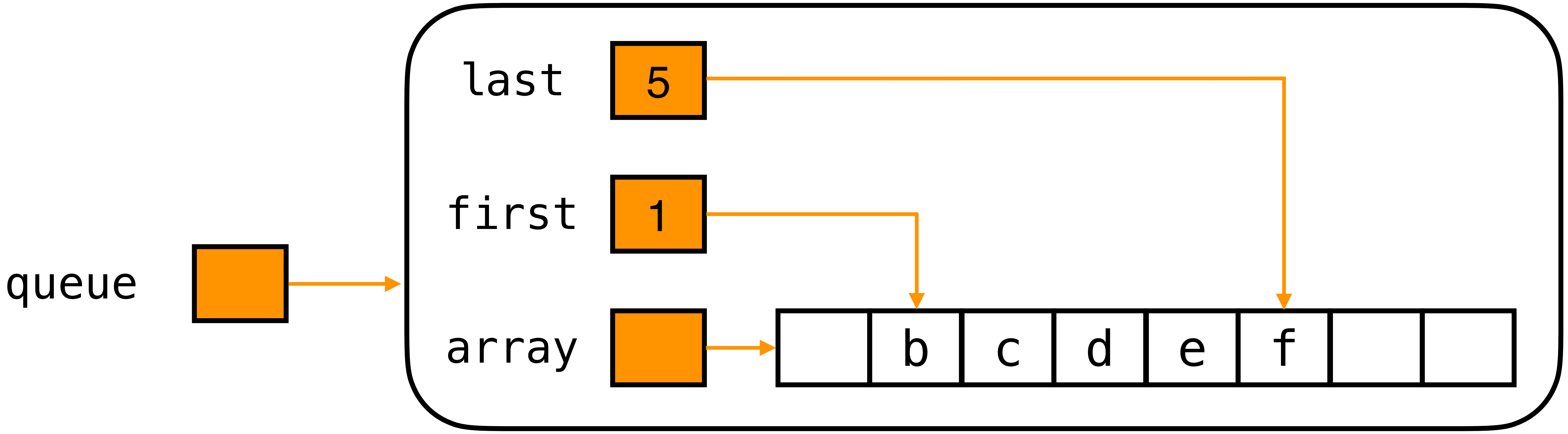
- When removing an element, **`first`** is moved one index to the right
- If **`first == last`** evaluates to **`true`**, the queue will be empty after **`dequeue()`** and we set **`first = last = -1`**

Example process: queue with an **`array`**



```
queue.enqueue(f);
```

Example process: queue with an **`array`**



- When adding an element, the value is inserted at **`array[++last]`**

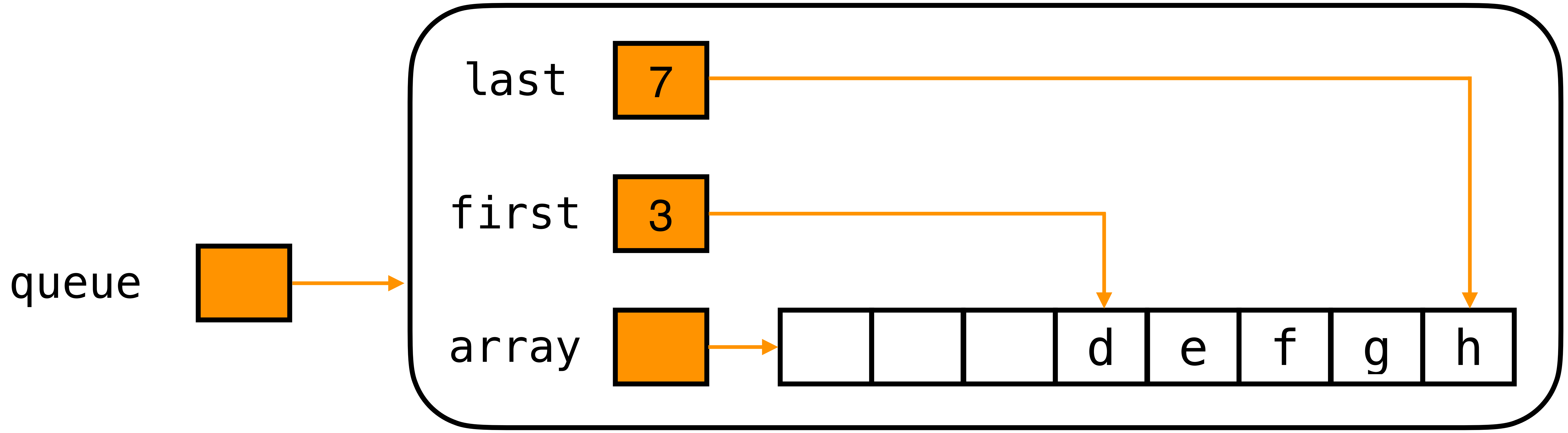
Queue: array boundaries for `enqueue()`

- What happens if `first` or `last` reach the end of the array so that `first == array.length` or `last == array.length` is `true`?
- **First idea:** double the array size (we also do this when the array is really full)
- **Disadvantage**: all elements in the left of `first` (if there are any) will never be able to be filled again

→ First reuse elements in the left of `first`

- If there is still space at the beginning of the array (i.e. from index 0), use these cells first: jump from `array.length – 1` to `0` (using the modulo operator `%`)
- A similar phenomenon with `dequeue()`: right shift of `first` when the array boundary is already reached

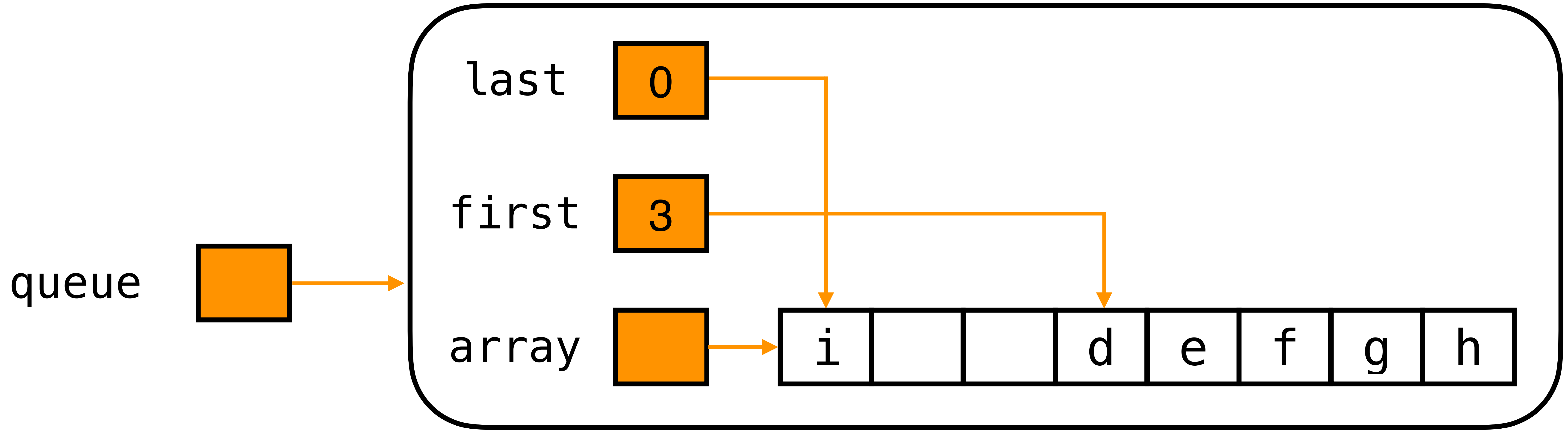
Example process: **`enqueue()`** reaching the end of the **`array`**



```
queue.enqueue(i);
```

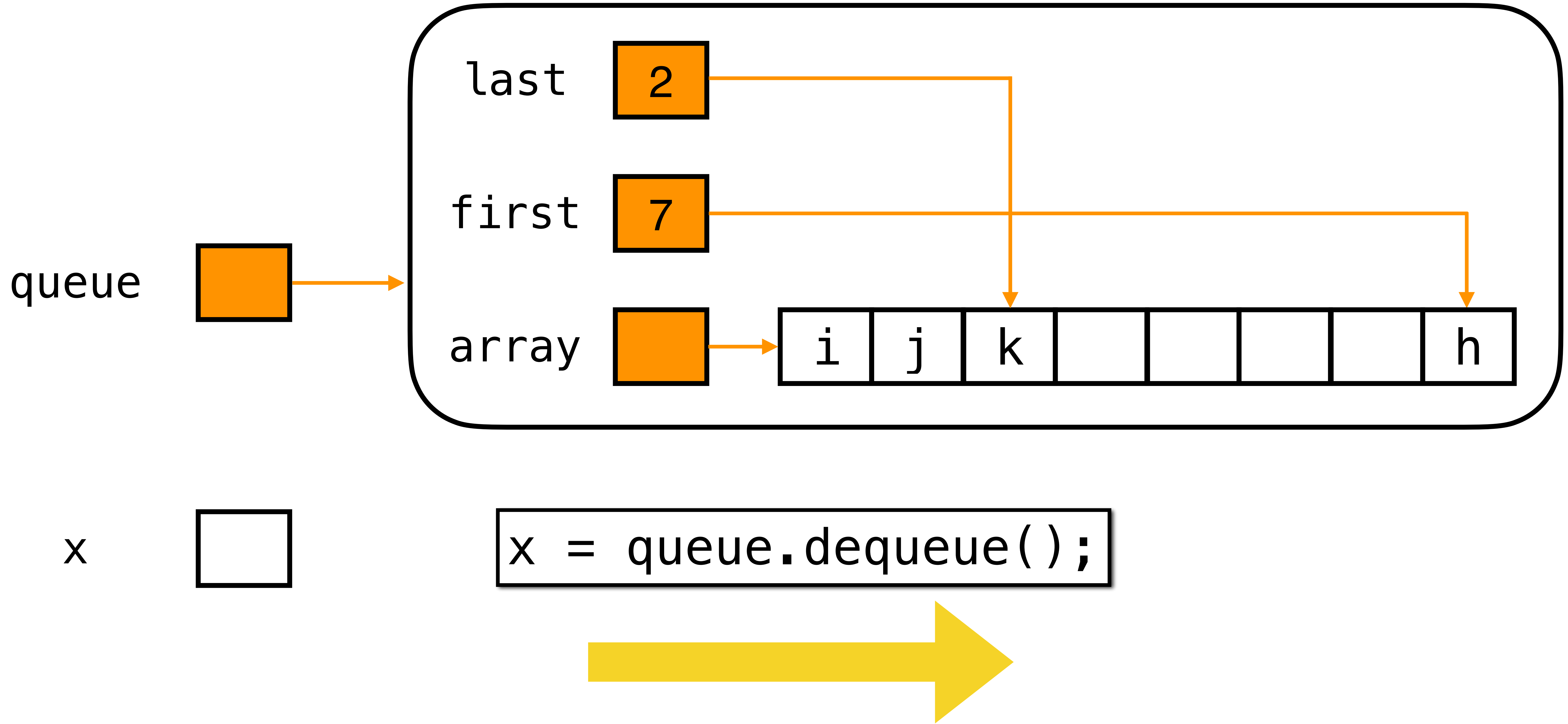
Assuming unused elements at the beginning of the array

Example process: **`enqueue()`** reaching the end of the **`array`**



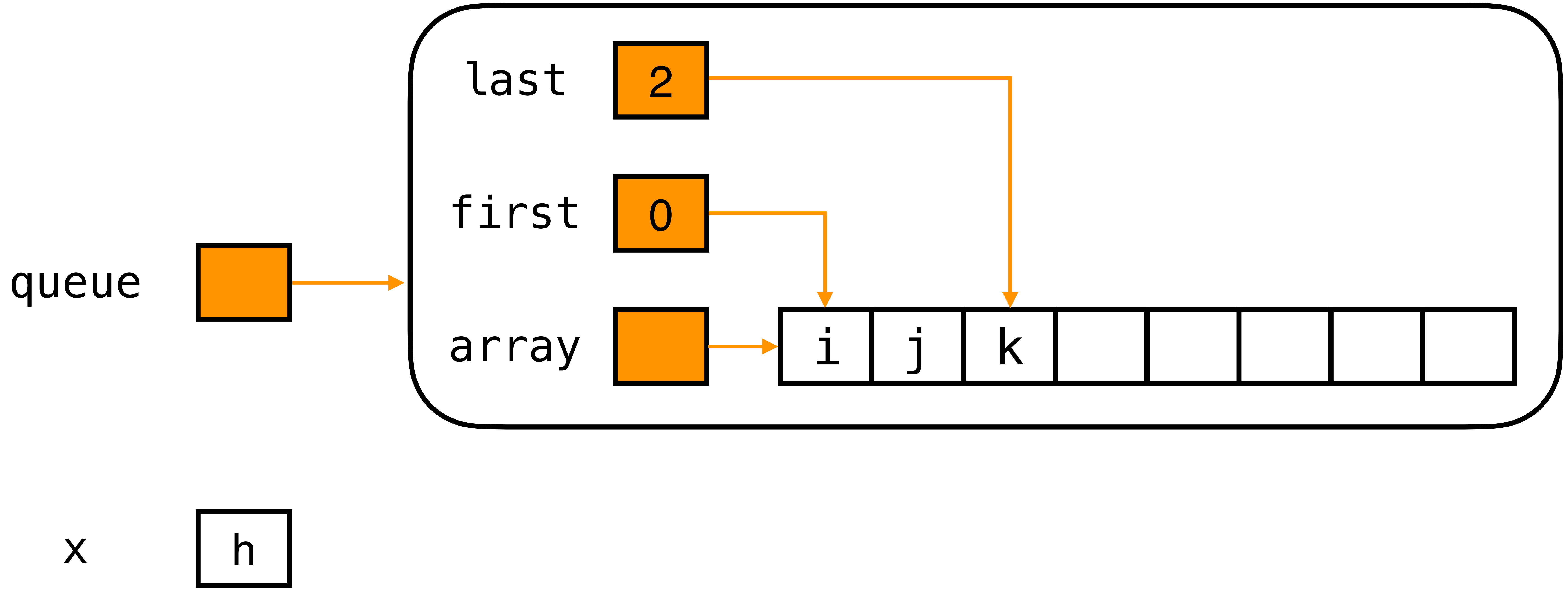
Assuming unused elements at the beginning of the array

Example process: **`dequeue()`** reaching the end of the **`array`**

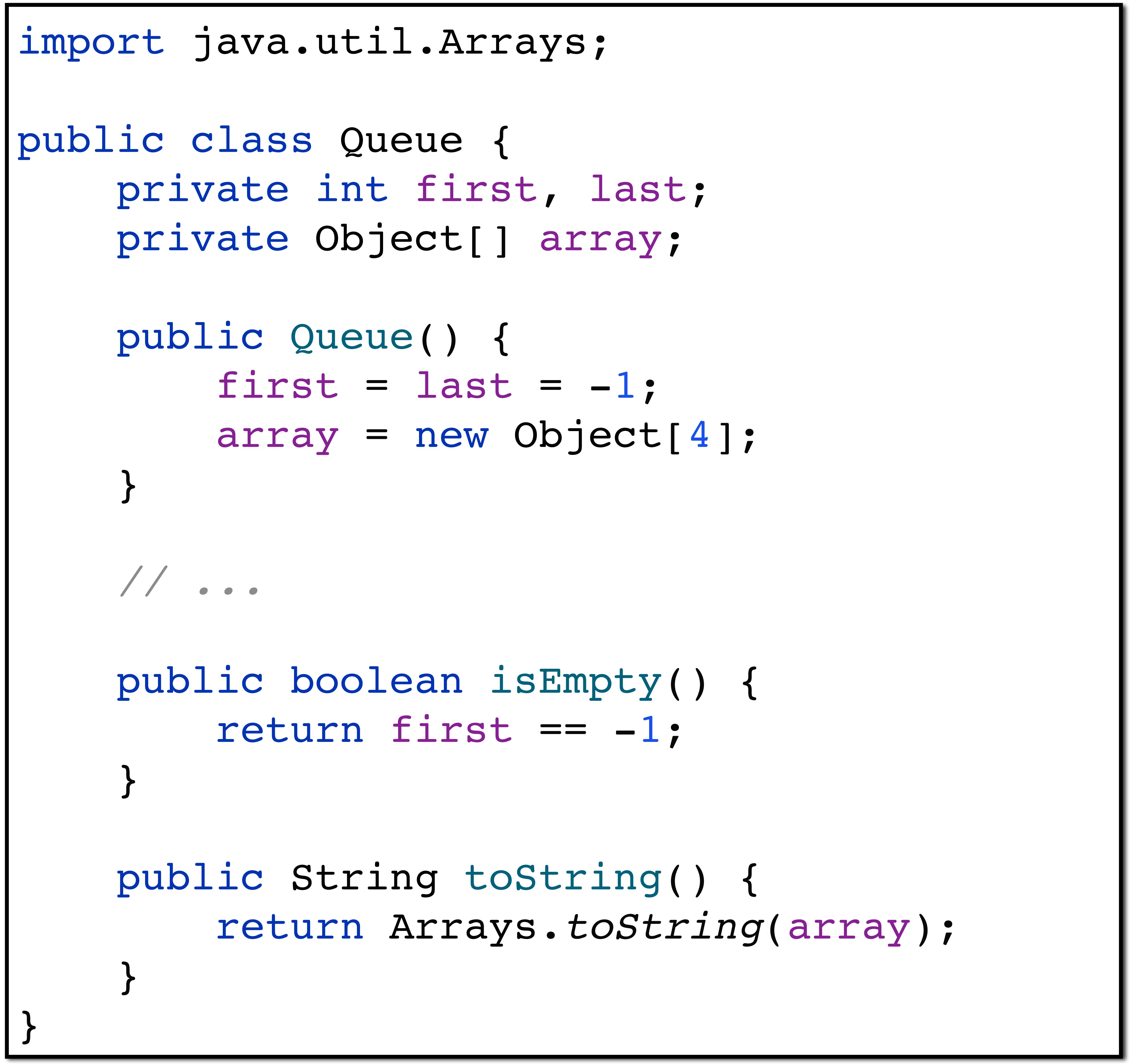


```
x = queue.dequeue();
```

Example process: **`dequeue()`** reaching the end of the **`array`**



Implementation: queue with an `array`



```java
import java.util.Arrays;

public class Queue {
    private int first, last;
    private Object[] array;

    public Queue() {
        first = last = -1;
        array = new Object[4];
    }

    // ...

    public boolean isEmpty() {
        return first == -1;
    }

    public String toString() {
        return Arrays.toString(array);
    }
}
```

Implementation: queue with an **`array`** - **`enqueue()`**

- If the queue is empty, **`first`** and **`last`** must be set to **`-1`**
- Otherwise, **`array`** is **full** exactly when **`element`** should be inserted at the position **`first`** (attention: we cannot assume **`first == 0`**)

`first` and **`(last + 1) % length`** would have the same value

- In this case we create a new array with double size
  We copy the elements

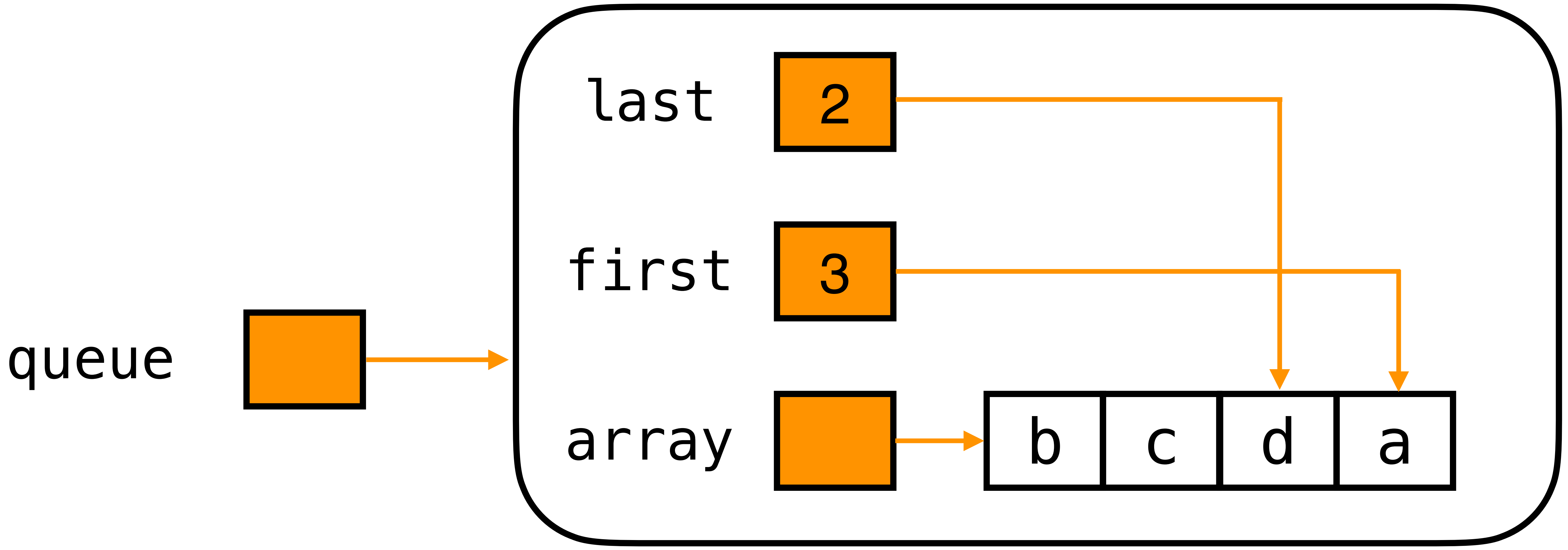
```
array[first], array[first + 1], ..., array[length - 1], array[0], array[1], ..., array[first - 1]
```

to

```
newArray[0],..., newArray[length - 1]
```

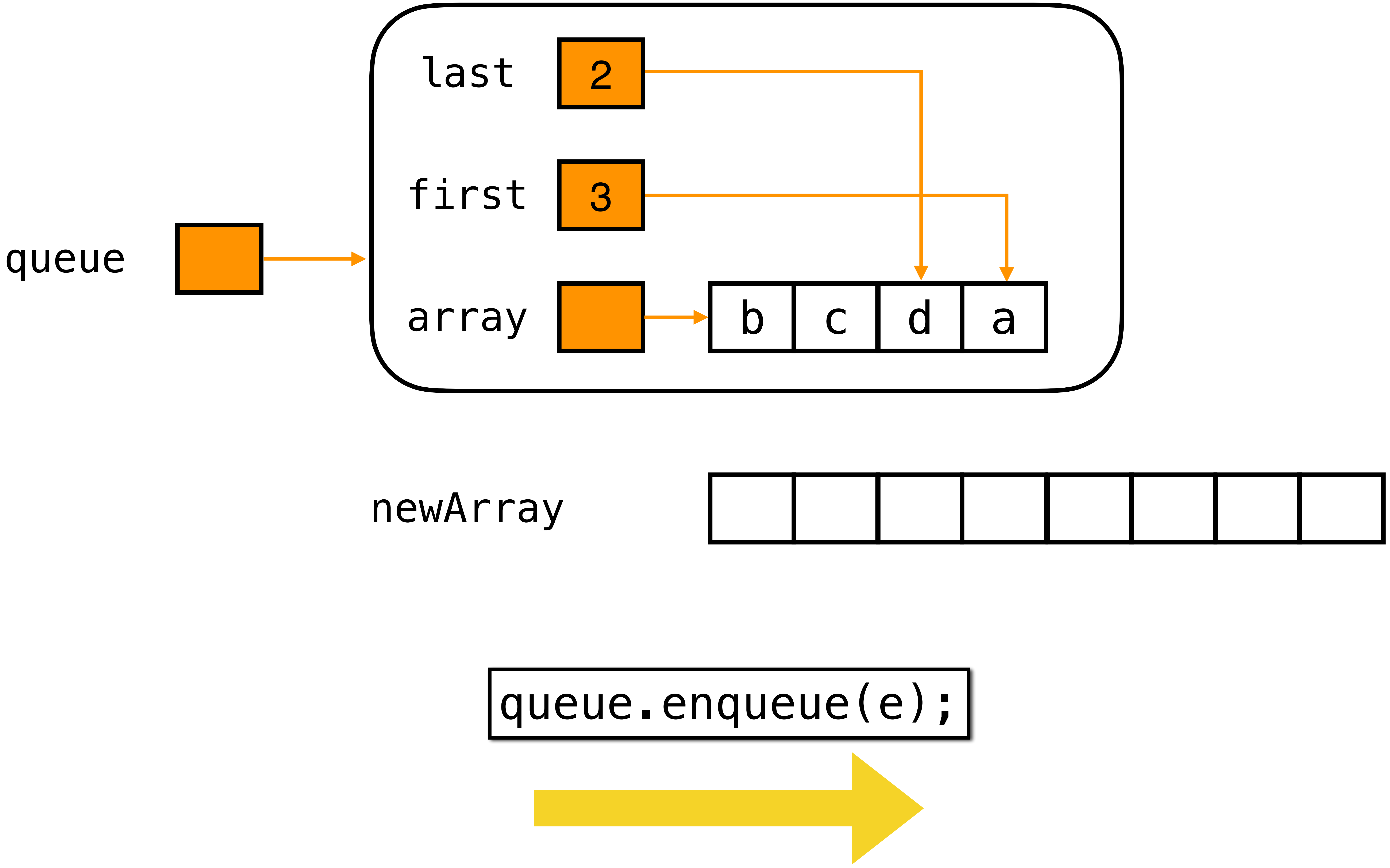
- Then `first = 0;` `last = array.length;` `array = newArray;`
- Now **`element`** can be stored at the position **`array[last]`**

Example process: doubling the **`array`** size



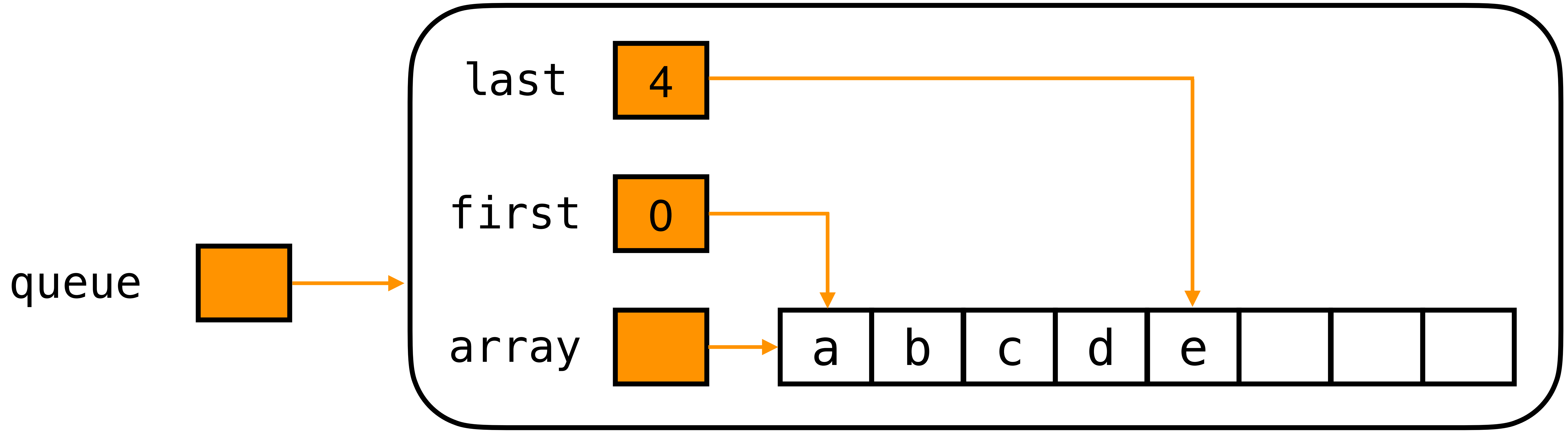
```
queue.enqueue(e);
```

Example process: doubling the array size

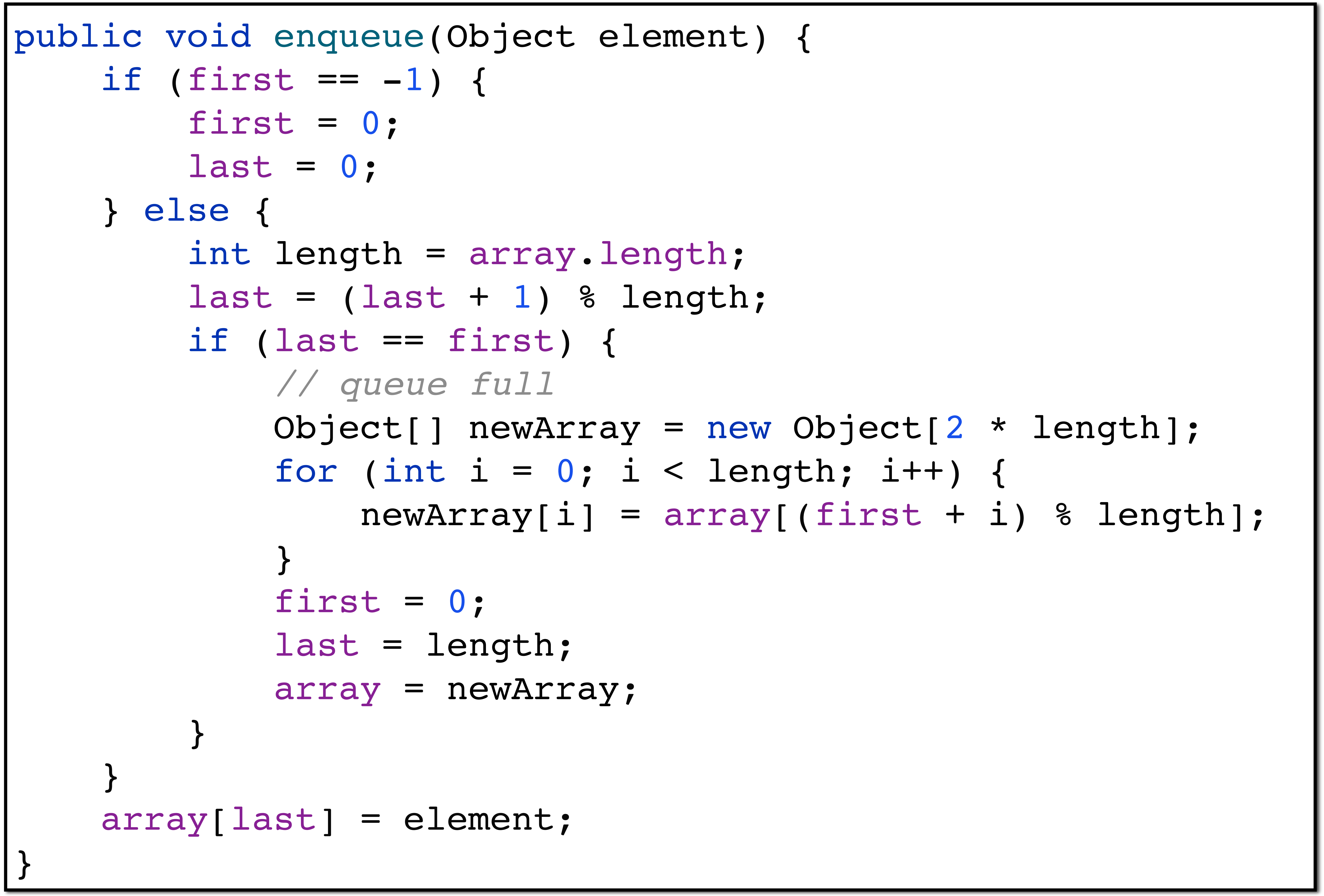


```
queue.enqueue(e);
```

Example process: doubling the **array** size



Implementation: queue with an **`array`** - **`enqueue()`**



```java
public void enqueue(Object element) {
    if (first == -1) {
        first = 0;
        last = 0;
    } else {
        int length = array.length;
        last = (last + 1) % length;
        if (last == first) {
            // queue full
            Object[] newArray = new Object[2 * length];
            for (int i = 0; i < length; i++) {
                newArray[i] = array[(first + i) % length];
            }
            first = 0;
            last = length;
            array = newArray;
        }
    }
    array[last] = element;
}
```

Implementation: queue with an **`array`** - **`dequeue()`**

- If after removing **`array[first]`** the queue is empty, **`first`** and **`last`** are set to **`-1`**
- Otherwise, **`first`** is incremented by 1 (modulo **`array.length`**)

```
public Object dequeue () {
    // assumption: first != -1
    Object result = array[first];
    if (first == last) {
        first = -1;
        last = -1;
    } else {
        first = (first + 1) % array.length;
    }
    return result;
}
```

Discussion

- In this implementation of **`dequeue()`**, the queue space is never reduced
- If the number of elements in the queue falls below a quarter of the length of **`array`**, we can replace it with another one with half the size (as with stacks)
- Attention: the elements in the queue do not need to be only at the beginning of **`array`**

Exercise

10 min

- Instantiate a **`queue`** (implemented using an **`array`**)
- Enqueue and dequeue multiple elements (e.g. Strings)
- Debug how **`enqueue`** and **`dequeue`** works to better understand these operations

Next steps

- **Tutor group** exercises
  - T03E01 - A carriage of line 6
  - T03E02 - Mia San FIFO
- **Homework** exercises
  - H03E01 - Call Me Maybe
  - H03E02 - Stack Track Voyager
- Read the following articles
  - https://www.digitalocean.com/community/tutorials/collections-in-java-tutorial
  - https://www.javatpoint.com/difference-between-array-and-arraylist

→ Due until **Wednesday, November 22, 13:00**

Summary

- The data type `List` is flexible and suited for rapid prototyping
- `LinkedList` vs. `ArrayList` vs. `Array`
- There are multiple implementations for useful data types `Stack (LIFO)` and `Queue (FIFO)`
- Often, there are additional operations for data types
- The **built-in Java collection types** (e.g. `List`, `Set`, `Map`) offer a great starting point and customizations for special use cases
  - Based on generic data types (type safety, no casting)
- The **enhanced for loop** allows to iterate through collection types (be careful with concurrent modification)

References

- https://www.digitalocean.com/community/tutorials/collections-in-java-tutorial
- https://www.educba.com/java-list-vs-array-list
- https://www.javatpoint.com/difference-between-array-and-arraylist
- https://www.baeldung.com/java-queue
- https://www.geeksforgeeks.org/stack-class-in-java

Break

10 min

The lecture will continue at **16:55**

Intermediate exam 1 information

- **Date:** Monday, 20 November 2023, 7:00 pm - 8:40 pm
- **Time**: 90 min + 10 min, **points**: 100
- **Content**: everything until the end of lecture week 02 (control structures)
- **Location**: Garching

You will receive an email with the actual lecture hall until Monday morning

- **Onsite**: you **must** participate in the assigned lecture hall
- **Setup**: use your own notebook
  - If you do **not** have a proper notebook, you can use a computer in the “Rechnerhalle” → fill out https://collab.dvb.bayern/x/8oHWDg until Thursday (Nov 16 evening)

In this case, make sure to try out the computer in the “Rechnerhalle" beforehand

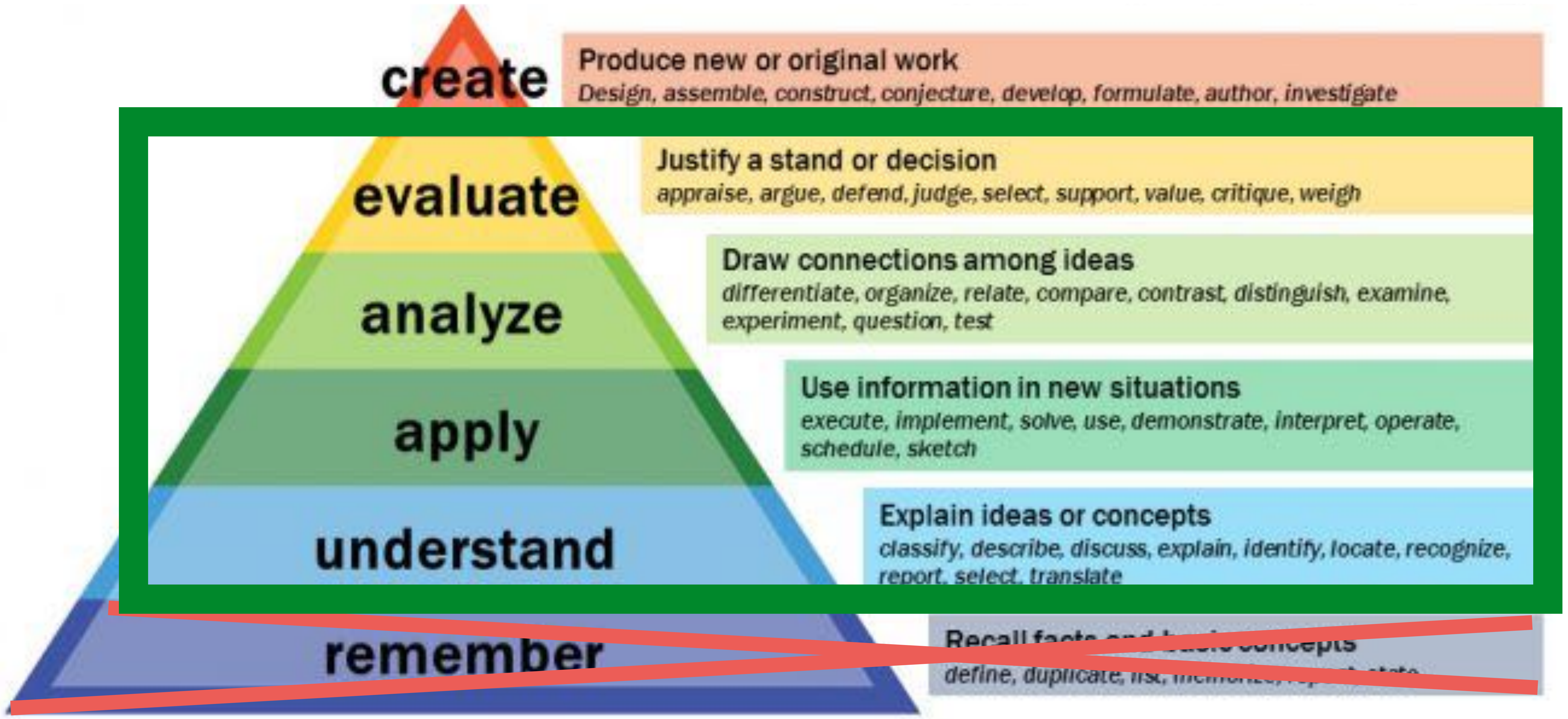
- **Open book**: use any resources (except AI)
- **Important: work alone, no communication is allowed!**

Artemis exam mode

- Exam mode features: https://artemis.cit.tum.de/features/students
  - Guide: https://docs.artemis.cit.tum.de/user/exams/students_guide
  - Tutorial: Artemis exam mode tutorial
- **Test exam**: try out the exam mode in Artemis and get familiar with it
  - **This afternoon (open until Sunday)**

General exam information

- The exercises focus on **understanding** and **problem solving**
- You **cannot** pass just with “learning by heart”
- Make sure that you are able to apply programming concepts to problem statements
- Review the learning goals of each lecture



create
Produce new or original work
Design, assemble, construct, conjecture, develop, formulate, author, investigate
evaluate
Justify a stand or decision
appraise, argue, defend, judge, select, support, value, critique, weigh
analyze
Draw connections among ideas
differentiate, organize, relate, compare, contrast, distinguish, examine, experiment, question, test
apply
Use information in new situations
execute, implement, solve, use, demonstrate, interpret, operate, schedule, sketch
understand
Explain ideas or concepts
classify, describe, discuss, explain, identify, locate, recognize, report, select, translate
remember
Recall facts and basic concepts
define, duplicate, list, memorize, repeat, state

Programming exercises

- Work on the programming exercise in your IDE
- Your code **must** compile on the build server
  - **Important**: compile failures will lead to 0 points
- No test feedback during the computer-based exam

Rules

- You must work on the exam **on your own**
  - Do **not** use chat applications (keep them closed all time)
  - Do **not** use artificial intelligence (OpenAI, ChatGPT, GitHub Copilot, or any similar systems are forbidden)
  - Do **not** post exam questions online
- You **must not** participate in the exam from home
- You may only use one monitor (no second monitor allowed)
- You must turn off all secondary devices (smartphone, tablet, etc.)
- Suspicious behavior, plagiarism and communication with other students is classified as cheating ("Unterschleif") and leads to consequences as mentioned in the APSO ("Allgemeine Prüfungs- und Studienordnung")
- In particular, the corresponding module in TUMonline will be marked as **failed (w. cheating)**

You **must** participate in the assigned lecture hall

Tips when using your own computer

- You are responsible for your computer: **before the exam**, make sure to install all required tools (browser, JDK 17, IntelliJ, git, etc.)
- Install all (operating system and application) updates before the exam: disable automatic updates for the duration of the exam
- **Close all windows and applications not needed for the exam**: this is especially important for all chat/communication applications
- Test your WiFi setup, and make sure you can connect to the different WiFi networks on campus: **eduroam** and **BayernWLAN**
- Keep the distractions to a minimum: disable notifications
- Check your git configuration for Artemis!
- Charge your battery
- Pack your laptop charger (and possibly an extension cord)
- Using a bluetooth mouse? Charge it

Technical issues during the exam

- If you experience technical issues, try to solve them on your own first (e.g. turn off WiFi and turn it on again, restart the computer, etc.)
- If you cannot solve the case on your own, raise your hand, a supervisor will try to help you
- In the unlikely case, the technical issue cannot be resolved (e.g. your computer breaks completely), you can resume the exam in the computer lab
  - In such a case, you will get additional time to compensate for the issue

Test exam

You have now the possibility to participate in a test exam on Artemis