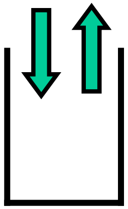
The Stack ADT

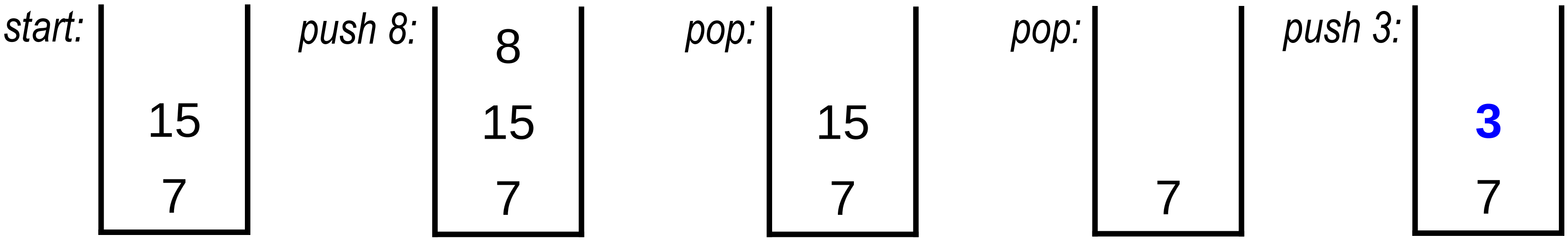
Computer Science 112
Boston University

Christine Papadakis-Kanaris

Stack ADT



- A stack is a sequence in which:
  - items can be added and removed only at one end (the *top*)
  - you can only access the item that is currently at the top
- Operations:
  - push: add an item to the top of the stack
  - pop: remove the item at the top of the stack
  - peek: get the item at the top of the stack, but don't remove it
  - isEmpty: test if the stack is empty
  - isFull: test if the stack is full
- Example: a stack of integers

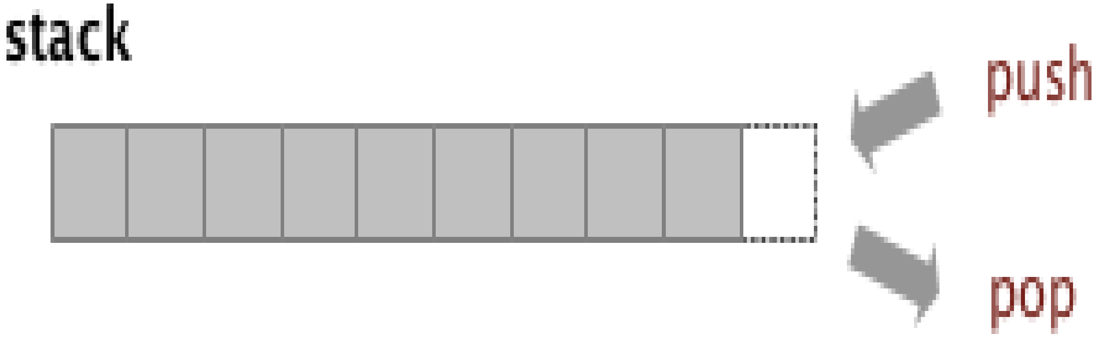


A Stack Interface: First Version

```
public interface Stack {
    boolean push(Object item);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}
```

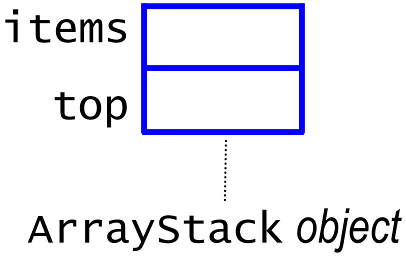
- `push()` returns `false` if the stack is full, and `true` otherwise.
- `pop()` and `peek()` take no arguments, because we know that we always access the item at the top of the stack.
  - return `null` if the stack is empty.
- The interface provides no way to access/insert/delete an item at an arbitrary position.
  - encapsulation allows us to ensure that our stacks are manipulated only in ways that are consistent with what it means to be stack

Array Implementation of a Stack



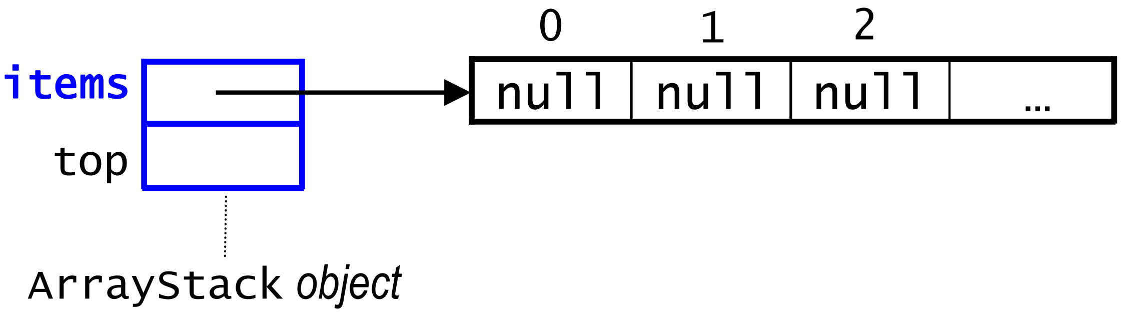
Implementing a Stack Using an Array: First Version

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item

    ...
}
```



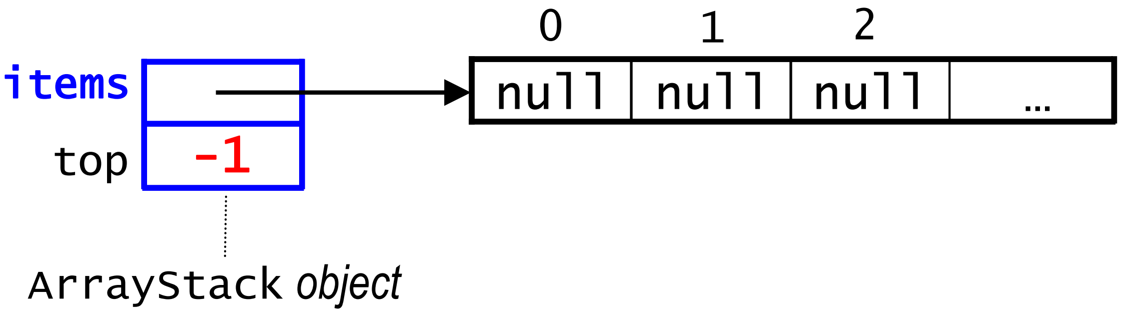
Implementing a Stack Using an Array: First Version

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item

    public ArrayStack(int maxSize) {
        items = new Object[maxSize];
        top = -1;
    }
    ...

}
```



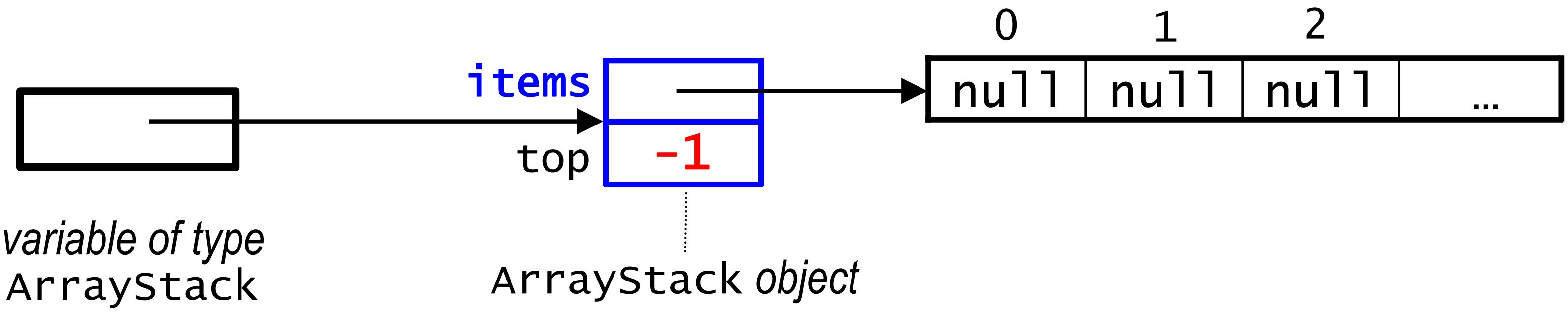
Implementing a Stack Using an Array: First Version

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item

    public ArrayStack(int maxSize) {
        items = new Object[maxSize];
        top = -1;
    }
    ...

}
```



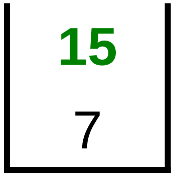
Implementing a Stack Using an Array: First Version

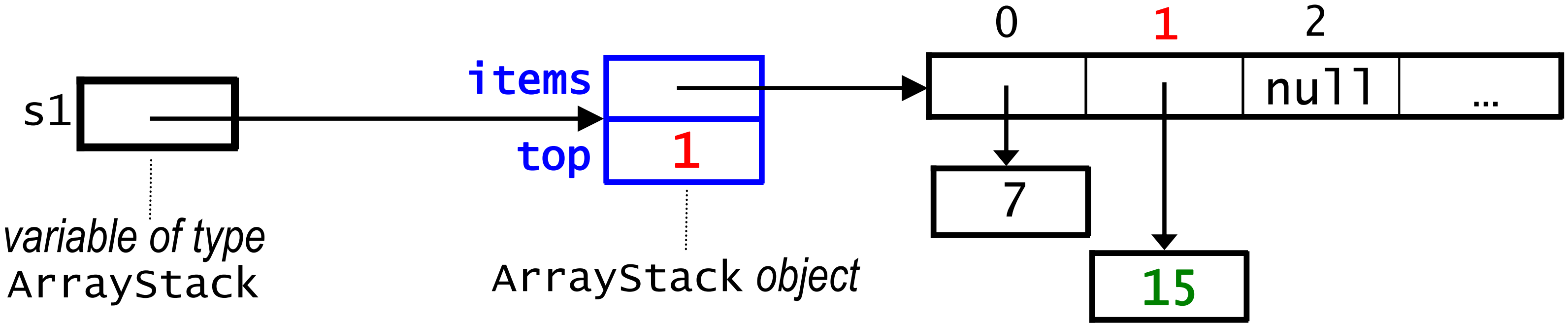
```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item

    public ArrayStack(int maxSize) {
        items = new Object[maxSize];
        top = -1;
    }
    ...

}
```



Implementing a Stack Using an Array: First Version

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item

    public ArrayStack(int maxSize) {
        items = new Object[maxSize];
        top = -1;
    }
    ...
}
```

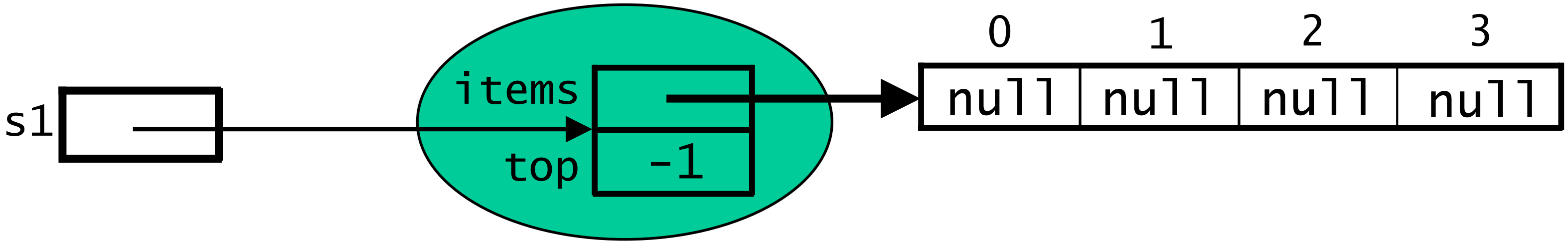
- Example: the stack 



- Items are added from left to right (top item = the rightmost one).
  - push() and pop() won't require any shifting!

Collection Classes and Data Types

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item
     ...
}
```

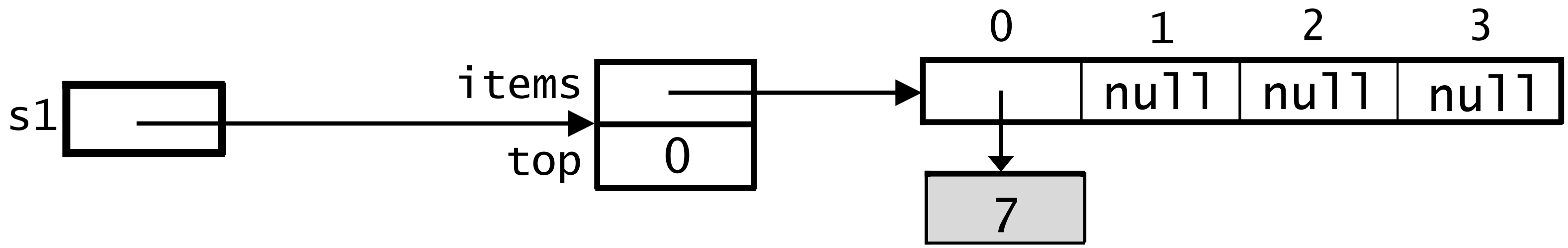


- So far, our collections have allowed us to add objects of any type.

```
ArrayStack s1 = new ArrayStack(4);
```

Collection Classes and Data Types

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item
     ...
}
```

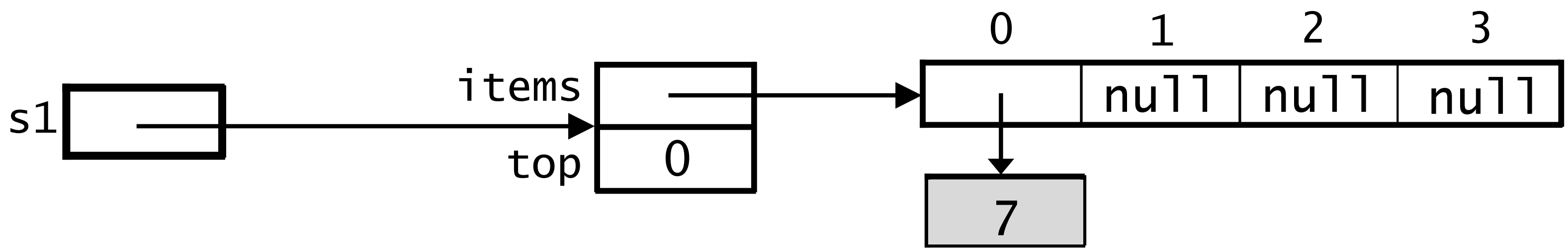


- So far, our collections have allowed us to add objects of any type.

```
ArrayStack s1 = new ArrayStack(4);
s1.push(7);    // 7 is turned into an Integer object for 7
```

Collection Classes and Data Types

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item
     ...
}
```



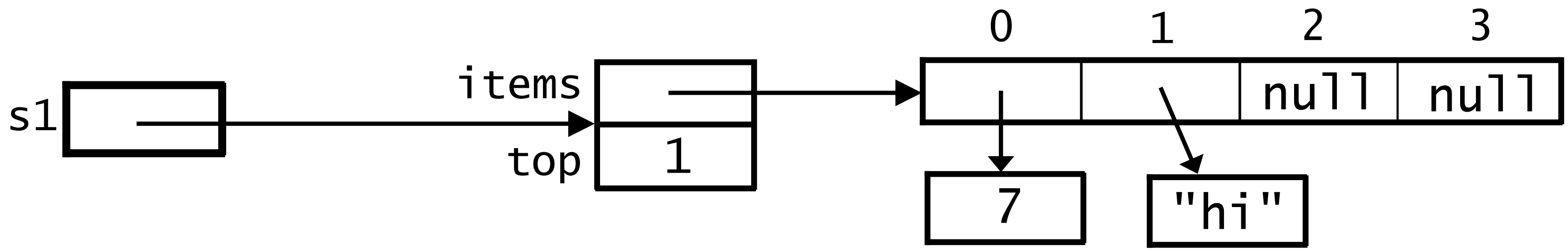
- So far, our collections have allowed us to add objects of any type.

```
ArrayStack s1 = new ArrayStack(4);
s1.push(7);    // 7 is turned into an Integer object for 7
```

An example of auto boxing. Java automatically creates a reference type from a primitive when needed.

Collection Classes and Data Types

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item
     ...
}
```

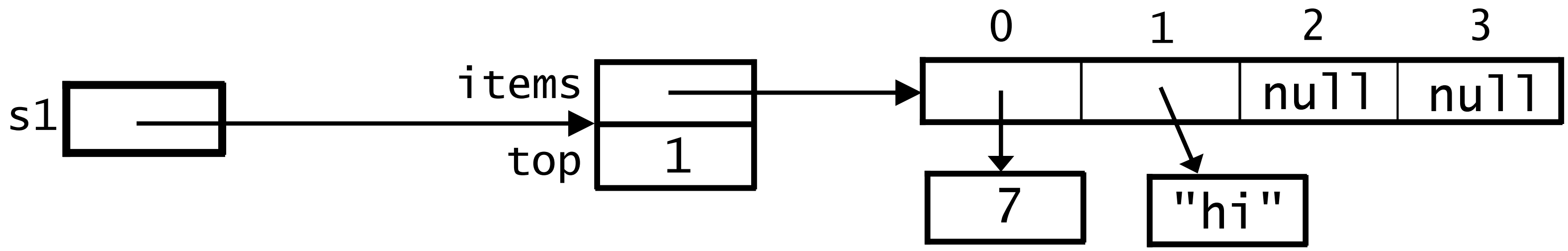


- So far, our collections have allowed us to add objects of any type.

```
ArrayStack s1 = new ArrayStack(4);
s1.push(7);    // 7 is turned into an Integer object for 7
s1.push("hi");
String item = s1.pop();            // won't compile
```

Collection Classes and Data Types

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item
     ...
}
```

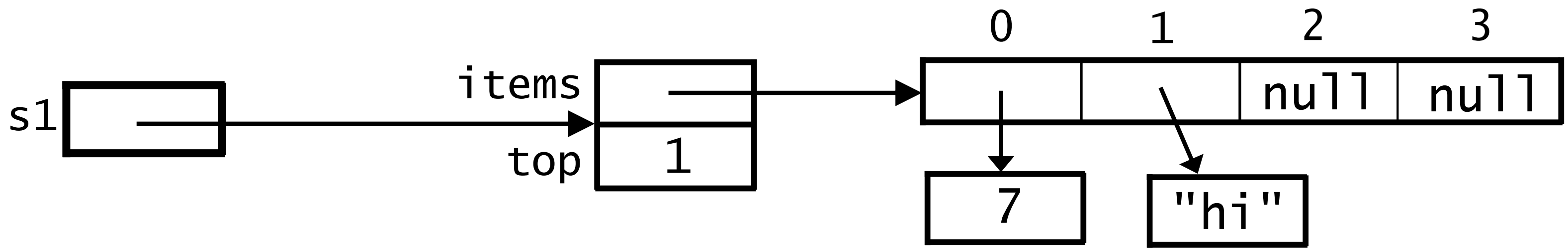


- So far, our collections have allowed us to add objects of any type.

```
ArrayStack s1 = new ArrayStack(4);
s1.push(7);    // 7 is turned into an Integer object for 7
s1.push("hi");
String item = s1.pop();           // won't compile
String item = (String) s1.pop();  // need a type cast
```

Collection Classes and Data Types

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item
     ...
}
```



- So far, our collections have allowed us to add objects of any type.

```
ArrayStack s1 = new ArrayStack(4);
s1.push(7);    // 7 is turned into an Integer object for 7
s1.push("hi");
String item = s1.pop();              // won't compile
String item = (String) s1.pop();  // need a type cast
```

- We'd like to be able to limit a given collection to one type.

```
ArrayStack<String> s2 = new ArrayStack<String>(10);
s2.push(7);                    // won't compile
s2.push("hello");
String item = s2.pop();    // no cast needed!
```

Limiting a Stack to Objects of a Given Type

- How about an interface for a stack of **strings**?

```
public interface StackInteger {
    boolean push(Integer item);
    Integer pop();
    Integer peek();
    boolean isEmpty();
    boolean isFull();
}
```

Limiting a Stack to Objects of a Given Type

- An interface for a stack of **strings**.

```
public interface StackString {
    boolean push(String item);
    String pop();
    String peek();
    boolean isEmpty();
    boolean isFull();
}
```

Do we really want to provide an interface for every possible type of data we want our stack to contain?

Limiting a Stack to Objects of a Given Type

- A *generic* interface and class.

```
public interface Stack<T> {
    boolean push(Object item);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}
```

- It includes a *type variable* `T` in its header and body.
    - used as a placeholder for the actual type of the items

Limiting a Stack to Objects of a Given Type

- A *generic* interface and class.
- Here's a generic version of our `Stack` interface:

```
public interface Stack<T> {
    boolean push(T item);
    T pop();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- It includes a *type variable* `T` in its header and body.
  - used as a placeholder for the actual type of the items

Implementing a Stack Using an Array: **First Version**

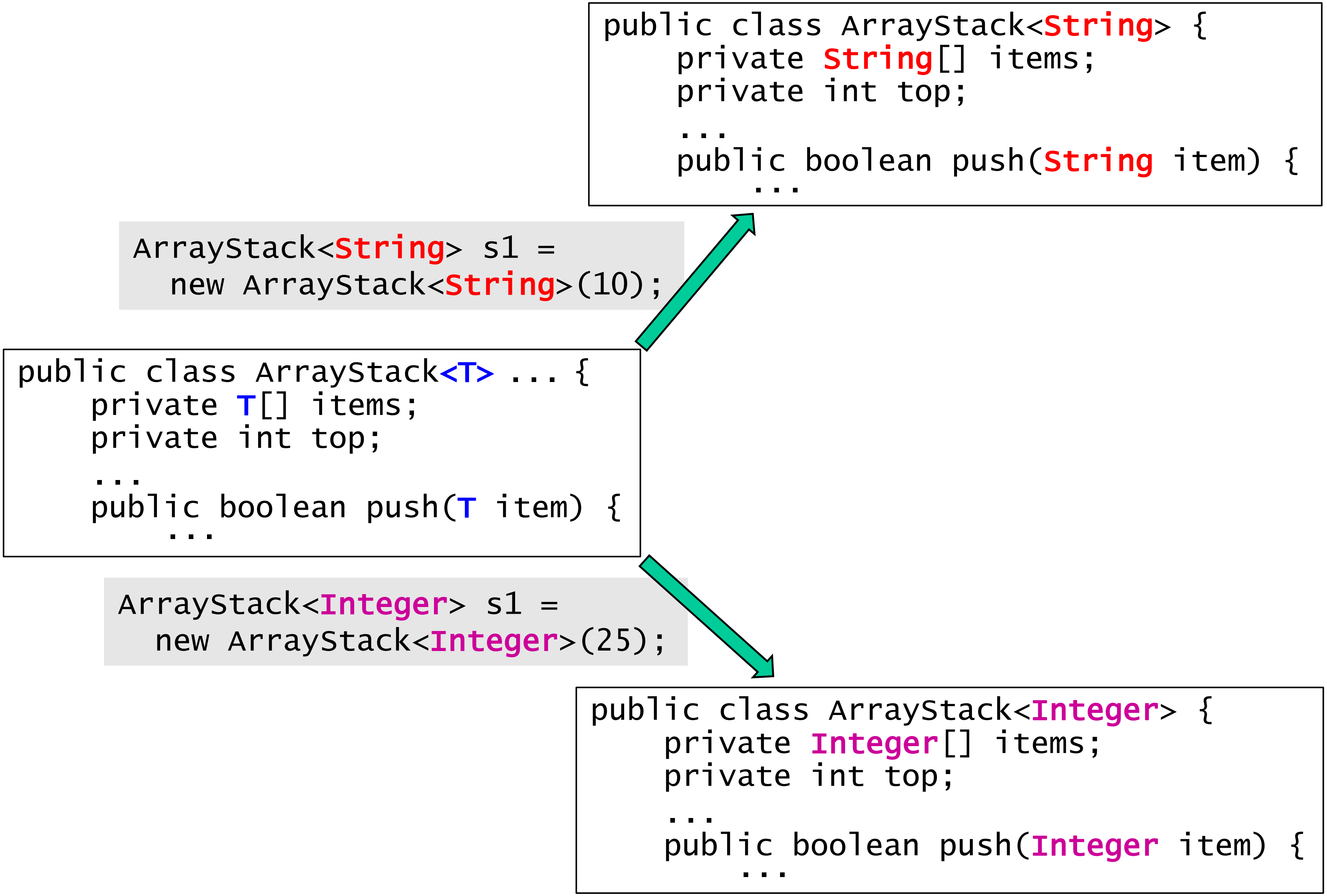
```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item

    ...
}
```

A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable T is used as a placeholder for the actual type of the items.

Using a Generic Class



```
public class ArrayStack<T> ... {
    private T[] items;
    private int top;
    ...
    public boolean push(T item) {
        ...
```

```
ArrayStack<String> s1 =
  new ArrayStack<String>(10);
```

```
public class ArrayStack<String> {
    private String[] items;
    private int top;
    ...
    public boolean push(String item) {
        ...
```

```
ArrayStack<Integer> s1 =
  new ArrayStack<Integer>(25);
```

```
public class ArrayStack<Integer> {
    private Integer[] items;
    private int top;
    ...
    public boolean push(Integer item) {
        ...
```

A Generic `ArrayStack` Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable `T` is used as a placeholder for the actual type of the items.

A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;     //   dex of the top item
    …
    public boolean push(T obj   ) {
        …
    }
    …
}
```

- Once again, a type variable T ne actual type of the items.

Note the use of the < > brackets in the name of the class we are creating and …

A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable T is … the actual type of the items.

… to specify the name of the interface that we are implementing but…

A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable T is u d as pl e actual type of the items.

*… we just use the place holder **T** when we need to specify the data type of the item.*

A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable `T` is used as a placeholder for the actual type of the items.

- When we create an `ArrayStack`, we specify the type of items that we intend to store in the stack:

```
ArrayStack<Integer> s1 = new ArrayStack<Integer>(10);
ArrayStack<String> s2 = new ArrayStack<String>(5);
```

- We can still allow for a mixed-type collection:

```
ArrayStack<Object> s3 = new ArrayStack<Object>(20);
```

`ArrayStack` Constructor

- Java doesn’t allow you to create an object or array using a type variable. Thus, we *cannot* do this:

```
public ArrayStack(int maxSize) {
    items = new T[maxSize];    // not allowed
    top = -1;
}
```

- To get around this limitation, we create an array of type `Object` and cast it to be an array of type `T`:

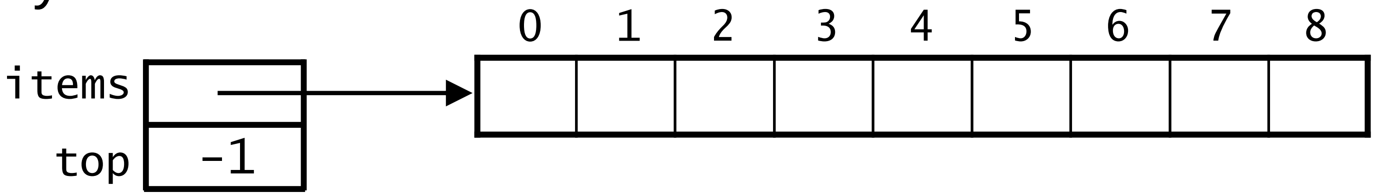
```
public ArrayStack(int maxSize) {
    items = (T[])new Object[maxSize];
    top = -1;
}
```

- The cast generates a compile-time warning, but we’ll ignore it.

- Java’s built-in `ArrayList` class takes this same approach.

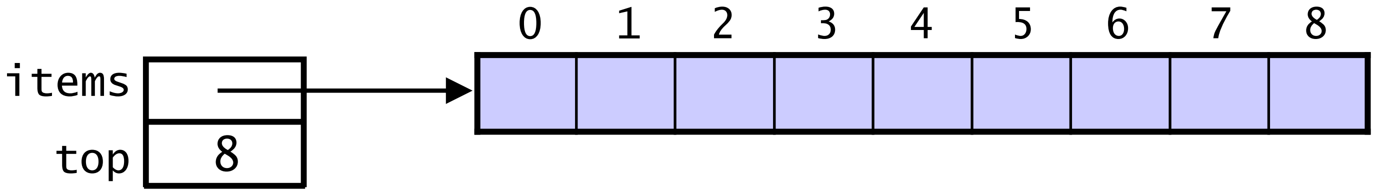
Testing if an `ArrayStack` is Empty or Full

- Empty stack:



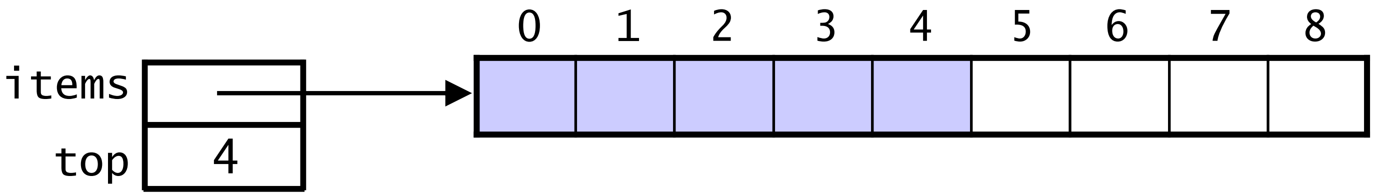
```
public boolean isEmpty() {
    return (top == -1);
}
```

- Full stack:



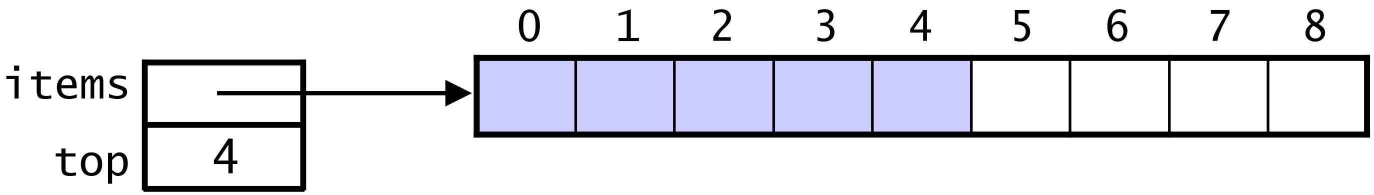
```
public boolean isFull() {
    return (top == items.length - 1);
}
```

Pushing an Item onto an `ArrayStack`



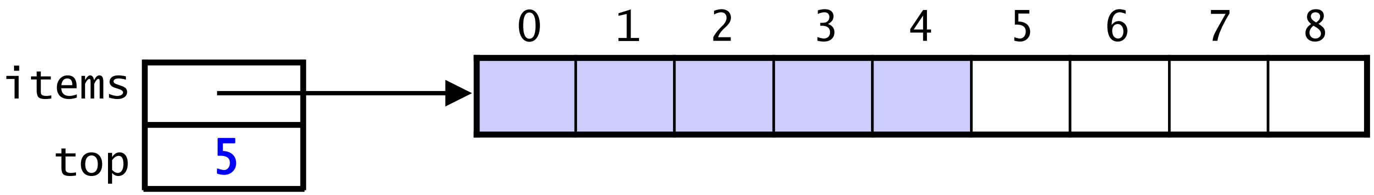
```
public boolean push(T item) {
    if (isFull()) {
        return false;
    }
    top++;
    items[top] = item;
    return true;
}
```

Pushing an Item onto an ArrayStack



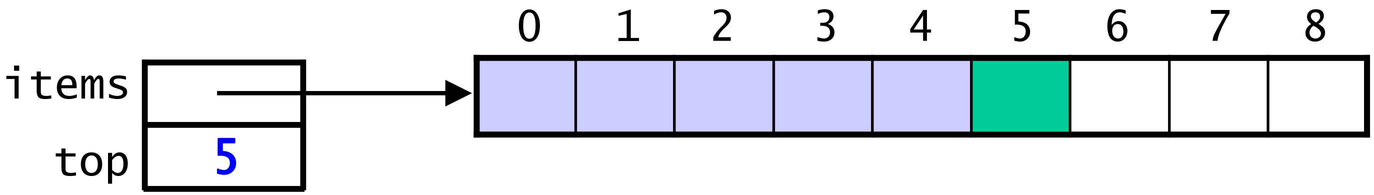
```
public boolean push(T item) {
    if (isFull()) {
        return false;
    }
    top++;
    items[top] = item;
    return true;
}
```

Pushing an Item onto an ArrayStack



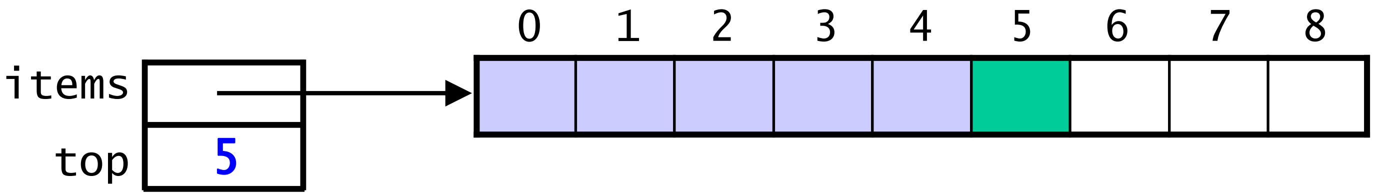
```
public boolean push(T item) {
    if (isFull()) {
        return false;
    }
    top++;
    items[top] = item;
    return true;
}
```

Pushing an Item onto an ArrayStack



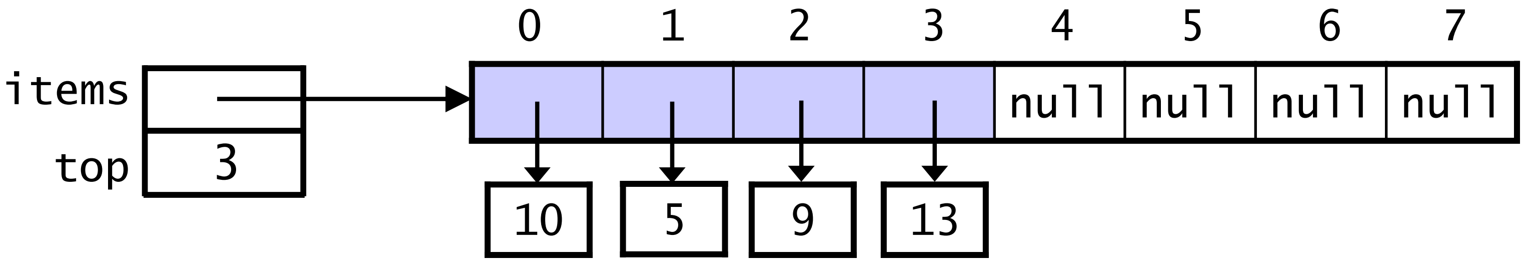
```
public boolean push(T item) {
    if (isFull()) {
        return false;
    }
    top++;
    items[top] = item;
    return true;
}
```

Pushing an Item onto an ArrayStack



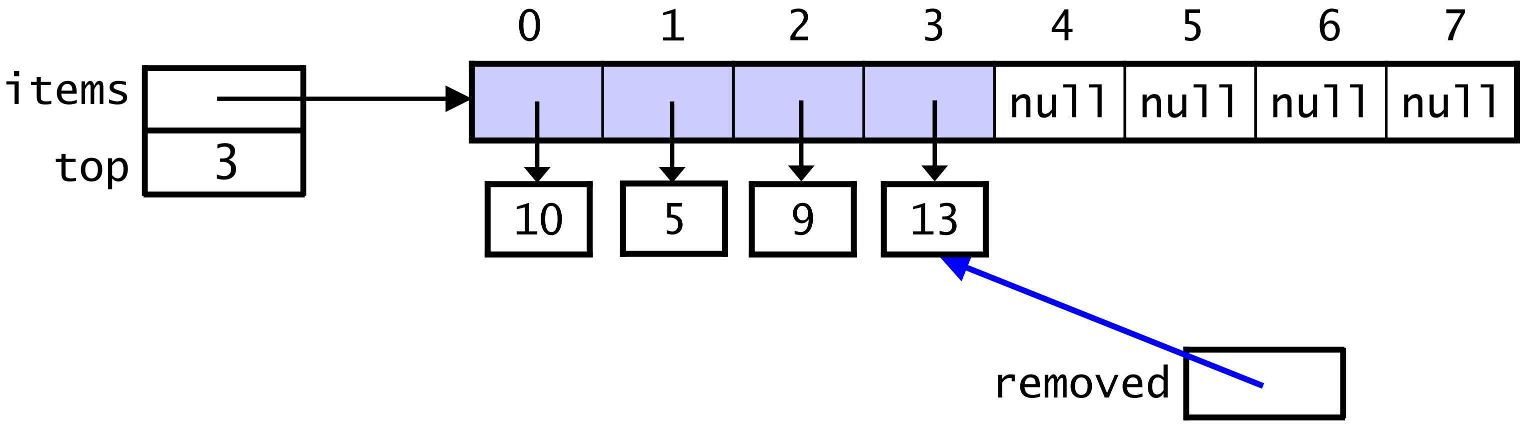
```
public boolean push(T item) {
    if (isFull()) {
        return false;
    }
    top++;
    items[top] = item;
    return true;
}
```

ArrayStack pop() and peek()



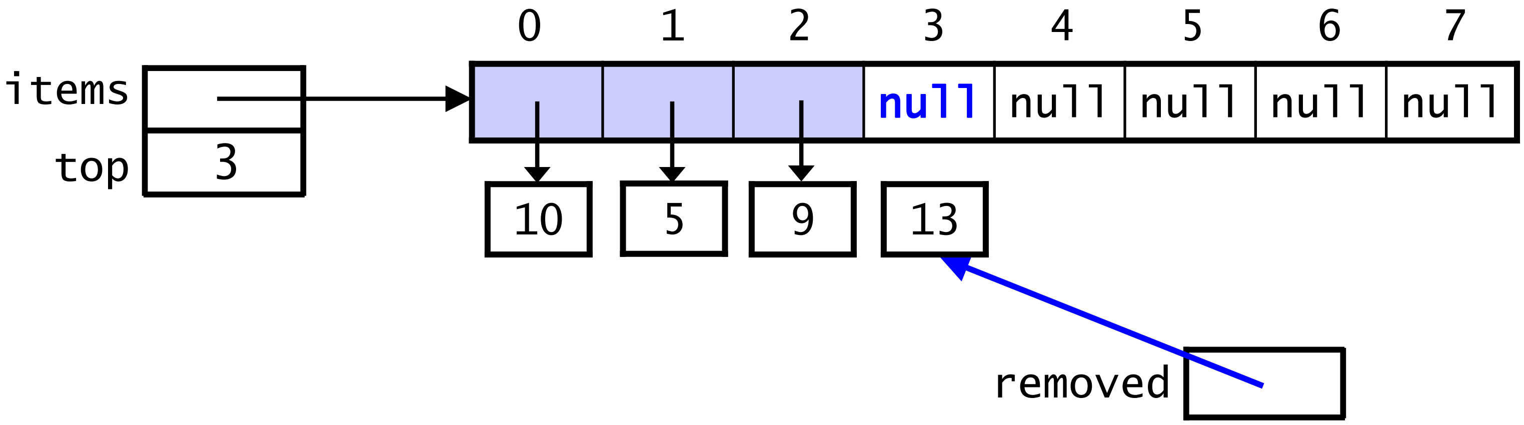
```
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

ArrayStack pop() and peek()



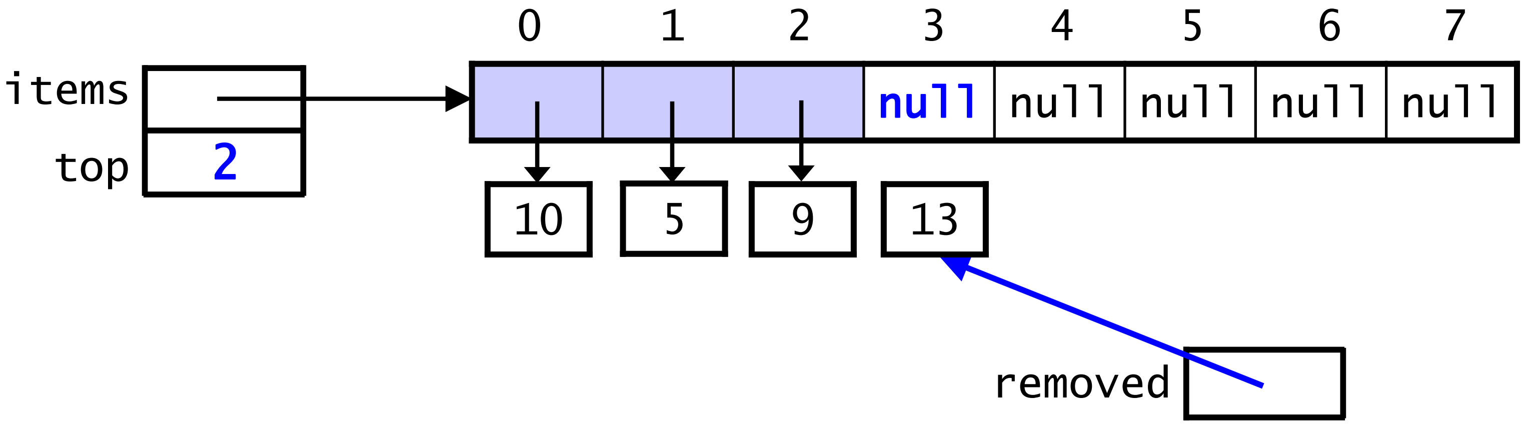
```
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

ArrayStack pop() and peek()



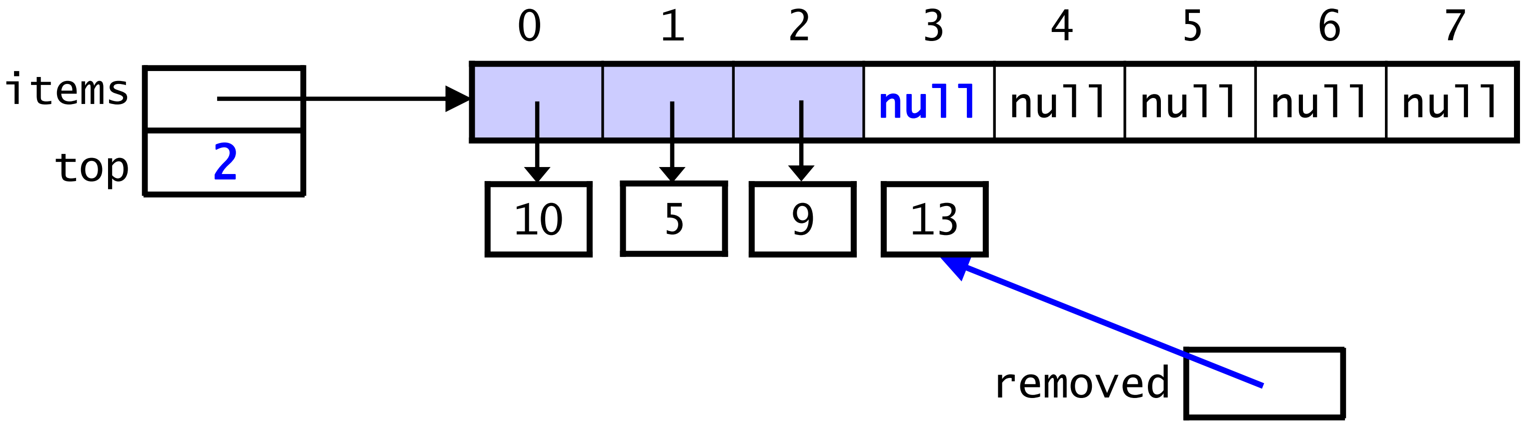
```
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

ArrayStack pop() and peek()



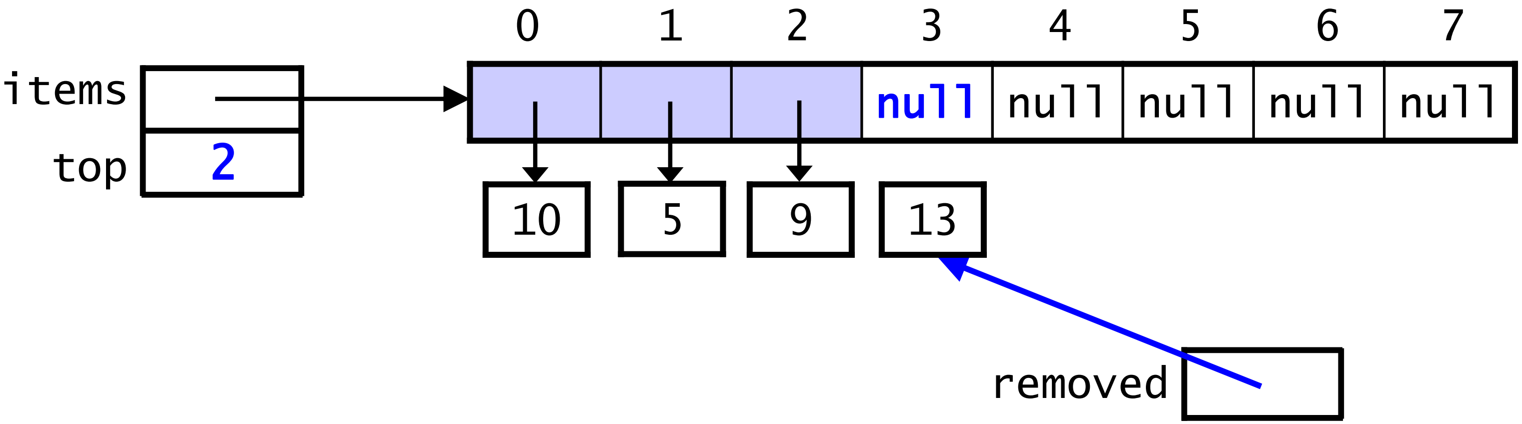
```
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

ArrayStack pop() and peek()



```
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

ArrayStack pop() and peek()



```
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

- peek just returns `items[top]` without decrementing `top`.