Programming Languages and Techniques (CIS120)

Lecture 24

Arrays, Java ASM

Chapter 21 and 22

Announcements

- HW6: Java Programming (Pennstagram)
  - Tuesday, November 5 at 11:59:59pm
- Reminder: please complete mid-semester survey
  - See post on Piazza
- Upcoming: Midterm 2
  - Friday, November 8th in class
  - Coverage: mutable state, queues, deques, GUI, Java

Design Exercise: Resizable Arrays

Arrays that grow without bound.

Step 1: Understand the Problem

- Fixed-size arrays are great, but...
  - What if you don't know when the array is created what size you need?
  - What if you want to add elements (mostly) to the end?
- Could use a linked list or queues, but...
  - You want mutability
  - and efficient array-indexing operations
- For simplicity: store only integer values*
- Note: Java Standard Library provides ArrayList
  - similar but significantly more expressive interface: "hybrid" linked list, array backing (also called a dynamic array)
  - often a good choice for this kind of task

*Java's implementation of generics turns out to interact badly with arrays, so we can't straightforwardly implement a generic version...

Step 2: Design the Interface

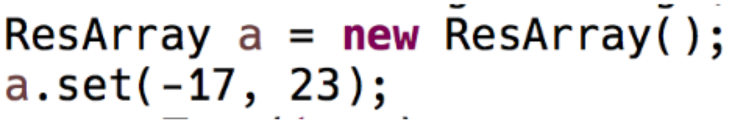
```
public class ResArray {

  /** Constructor, takes no arguments. */
  public ResArray() { … }

  /** Access the array at position i. If position i has not yet
   * been initialized, return 0.
   */
  public int get(int i) { … }

  /** Modify the array at position i to contain the value v. */
  public void set(int i, int v) { … }

  /** Return the extent of the array. */
  public int getExtent() { … }

  /** Return the array data (up to first 0) */
  public int[] values() { … }


}
```

Step 3: Write Test Cases

- Use JUnit to write tests of the interface behavior.
  - remember: `@Test` annotation
  - use assertion methods `AssertTrue, AssertFalse, AssertEquals`, etc.

- Questions to ask yourself:
  - How do the methods of the interface interact?
  - What properties do we expect them to satisfy?
  - What are the corner cases?

What behavior would you expect for this code?



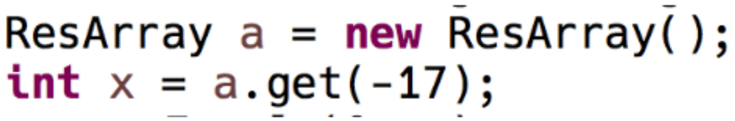
```
ResArray a = new ResArray();
a.set(-17, 23);
```

The code succeeds but does not modify the ResArray data.

The code fails with an ArrayIndexOutOfBoundsException

The code fails with an IllegalArgumentException

What behavior would you expect from the following code?



```
ResArray a = new ResArray();
int x = a.get(-17);
```

The code succeeds, resulting in x of value 0

The code succeeds, resulting in x of some non-zero value

The code fails with an ArrayIndexOutOfBounds exception

The code fails with an IllegalArgumentException

Which Behavior?

```
// test that an expected exception is raised
@Test(expected = IndexOutOfBoundsException.class)
public void testSetNegativeArg () {
    ResArray a = new ResArray();
    a.set(-17, 23);
    assertTrue(true);
}
```

- Which exception to throw? Maybe neither ArrayIndexOutOfBoundsException nor IllegalArgumentException...
- Neither is quite precise, so use IndexOutOfBoundsException
- Be consistent across get and set.
  - Inconsistent behavior leads to bugs...
- We'll see more when we get to Java Collections

Step 4: Implement the Behavior

- Implement the Behavior: See ResArray.java
  - What invariants to we maintain?
  - Does the code properly encapsulate those invariants?

See ResArrayTest.java and ResArray.java

DEMO

Adding *extent*

```
private int extent = 0;
  /* INVARIANT: extent = 1+index of last nonzero
   * element, or 0 if all elements are 0. */

/** Modify the array at position i to contain the value v. */
  public void set(int idx, int val) {
    if (idx < 0) {
      throw new IndexOutOfBoundsException();
    }
    grow(idx);
    data[idx] = val;
    if (val != 0 && idx+1 > extent) {
      extent = idx+1;
    }
    if (val == 0 && idx+1 == extent) {
      while (extent > 0 && data[extent-1] == 0) {
        extent--;
      }
    }
  }

  /** Return the extent of the array. */
  public int getExtent() {
    return extent;
  }
```

Object Invariant: extent is 1 past the last nonzero value in data (can be 0 if the array is all zeros)

Revenge of the Son of the Abstract Stack Machine

The Java Abstract Stack Machine

Objects, Arrays, and Static Methods

Java Abstract Stack Machine

- Similar to OCaml Abstract Stack Machine
  - Workspace
    - Contains the currently executing code
  - Stack
    - Remembers the values of local variables and "what to do next" after function/method calls
  - Heap
    - Stores reference types: objects and arrays

- Key differences:
  - Everything, including stack slots, is mutable by default
  - Objects store *what class was used to create them*
  - *Arrays store type information and length*
  - *New component: Class table (coming soon)*

Java Primitive Values

- The values of these data types occupy (less than) one machine word and are stored directly in the stack slots.

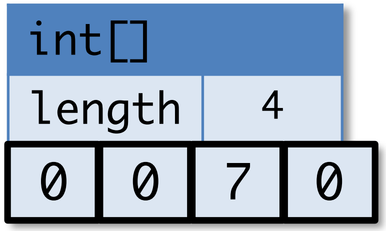
Type	Description	Values
`byte`	8-bit	`-128 to 127`
`short`	16-bit integer	`-32768 to 32767`
`int`	32-bit integer	-2^{31} to $2^{31} - 1$
`long`	64-bit integer	-2^{63} to $2^{63} - 1$
`float`	32-bit IEEE floating point	
`double`	64-bit IEEE floating point	
`boolean`	true or false	`true false`
`char`	16-bit unicode character	`'a' 'b' '\u0000'`

Heap Reference Values

Arrays

- Type of values that it stores
- Length
- Values for all of the array elements

```
int [] a =
    { 0, 0, 7, 0 };
```



int[]	
length	4

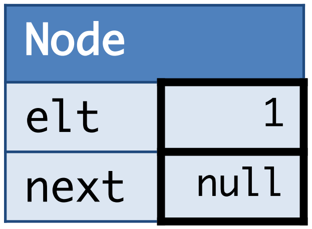
0	0	7	0

length *never* mutable; elements *always* mutable

Objects

- Name of the class that constructed it
- Values for all of the fields

```
class Node {
   private int elt;
   private Node next;
   …
}
```



Node	
elt	1
next	null

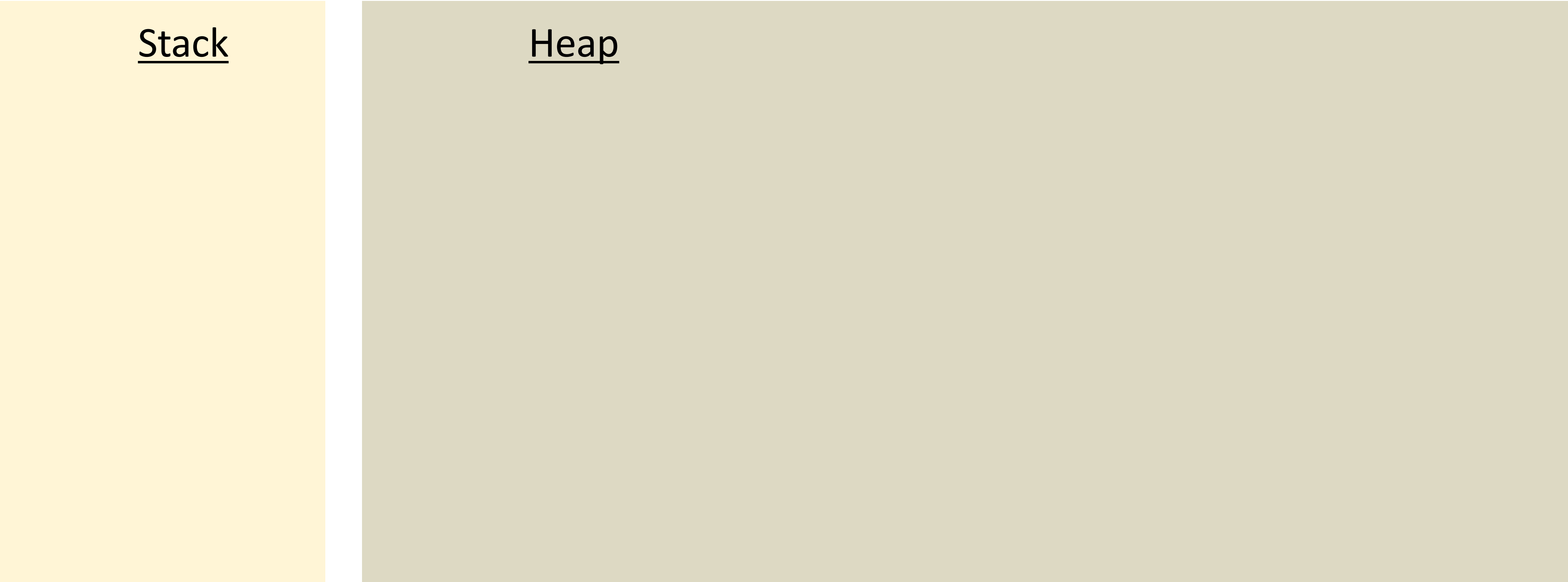
fields may or may not be mutable

public/private not tracked by ASM

ResArray ASM

Workspace

```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



Stack

Heap

ResArray ASM

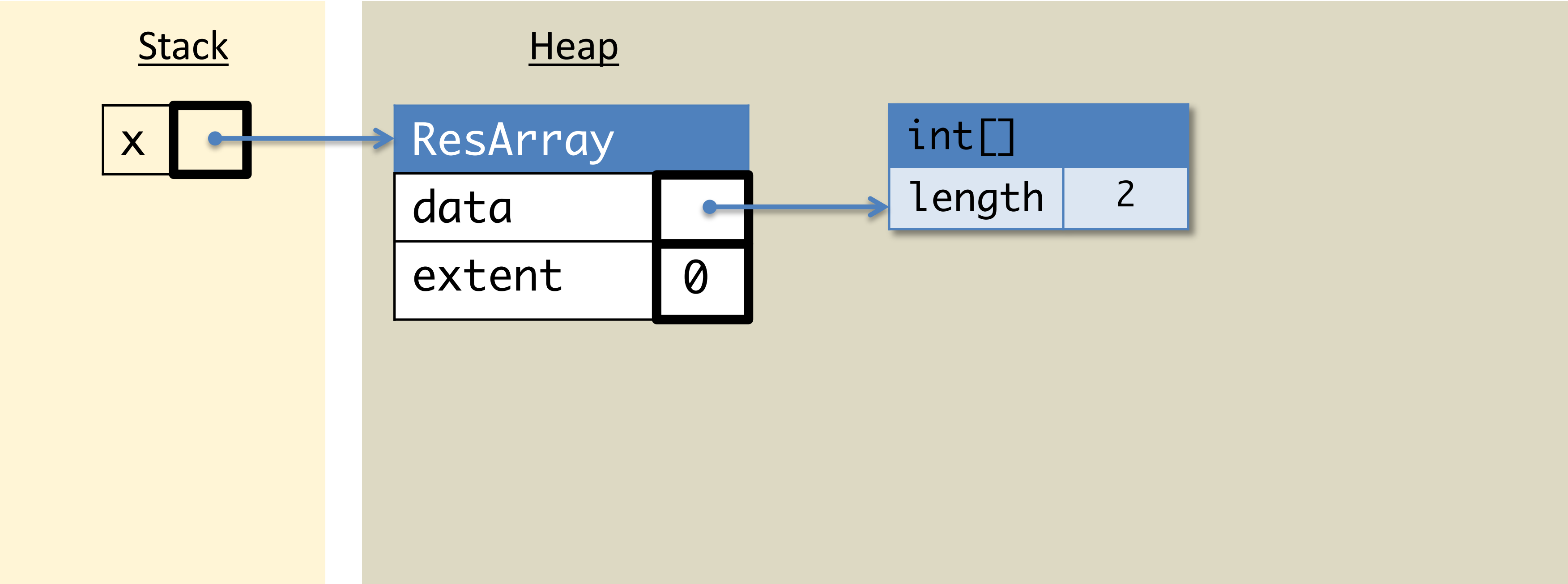
Workspace

~~ResArray x = new ResArray();~~

x.set(3,2);

x.set(4,1);

x.set(4,0);



ResArray ASM

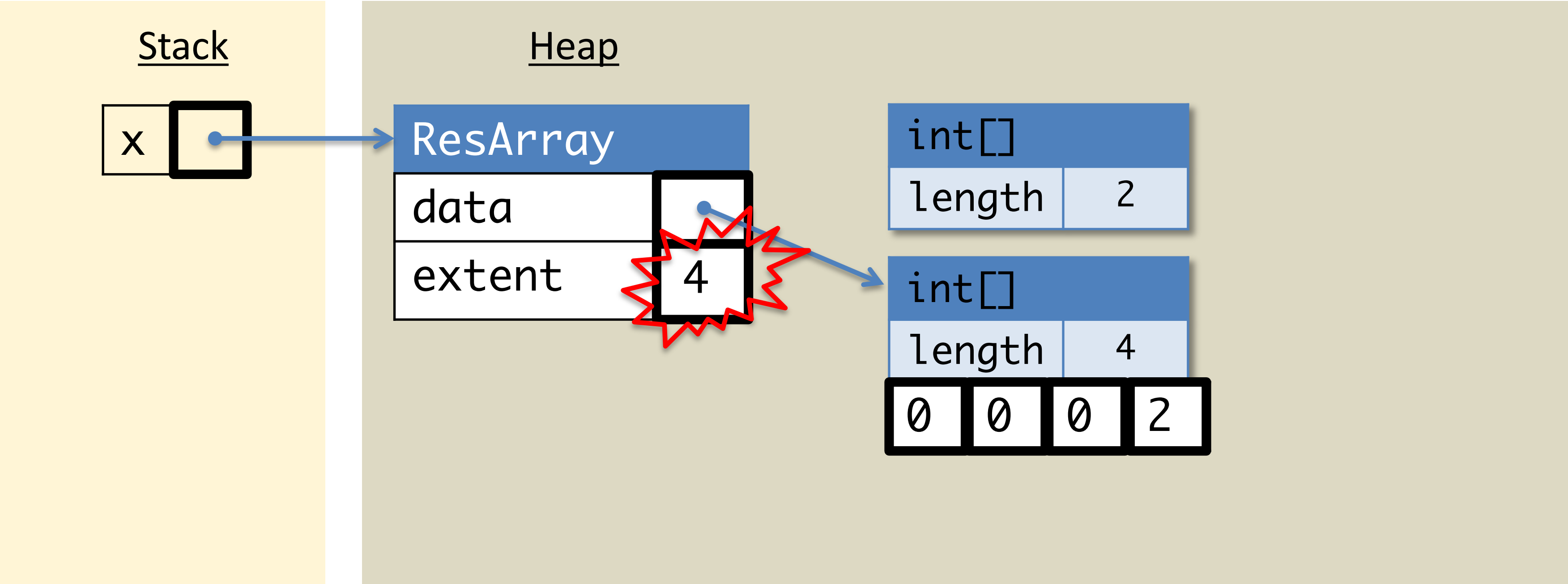
Workspace

~~ResArray x = new ResArray();~~

~~x.set(3,2);~~

x.set(4,1);

x.set(4,0);



ResArray ASM

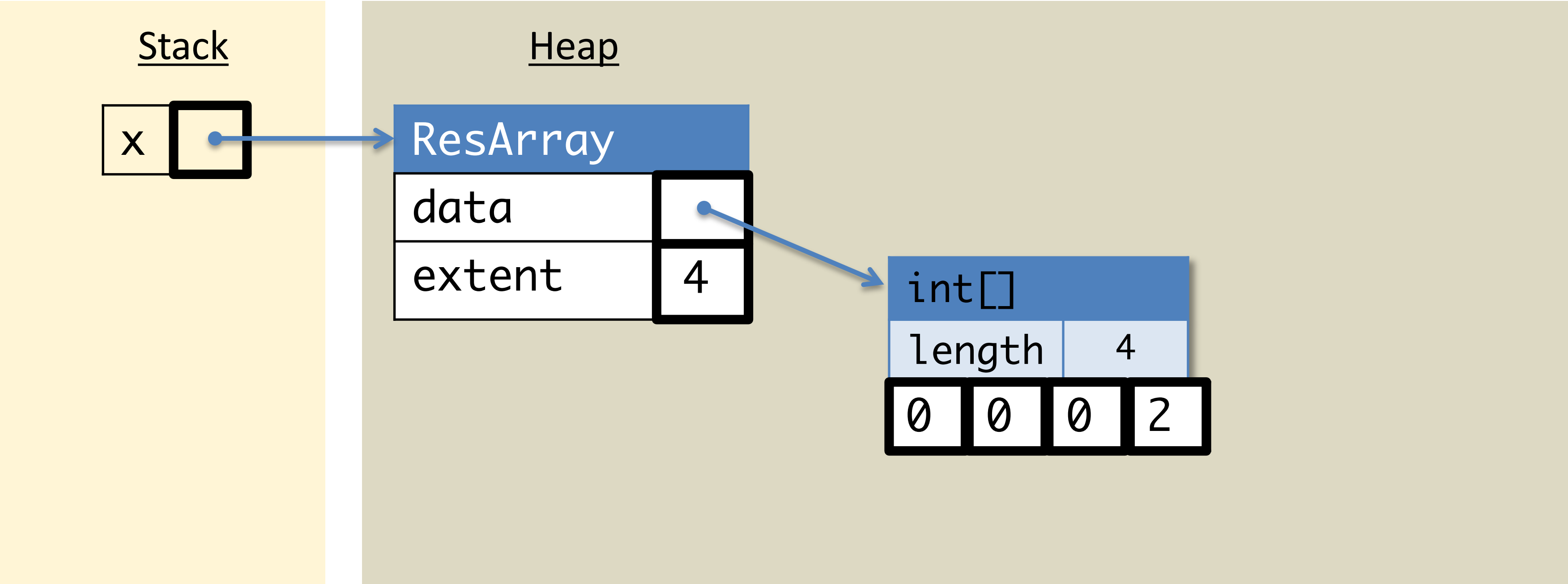
Workspace

~~ResArray x = new ResArray();~~

~~x.set(3,2);~~

x.set(4,1);

x.set(4,0);



ResArray ASM

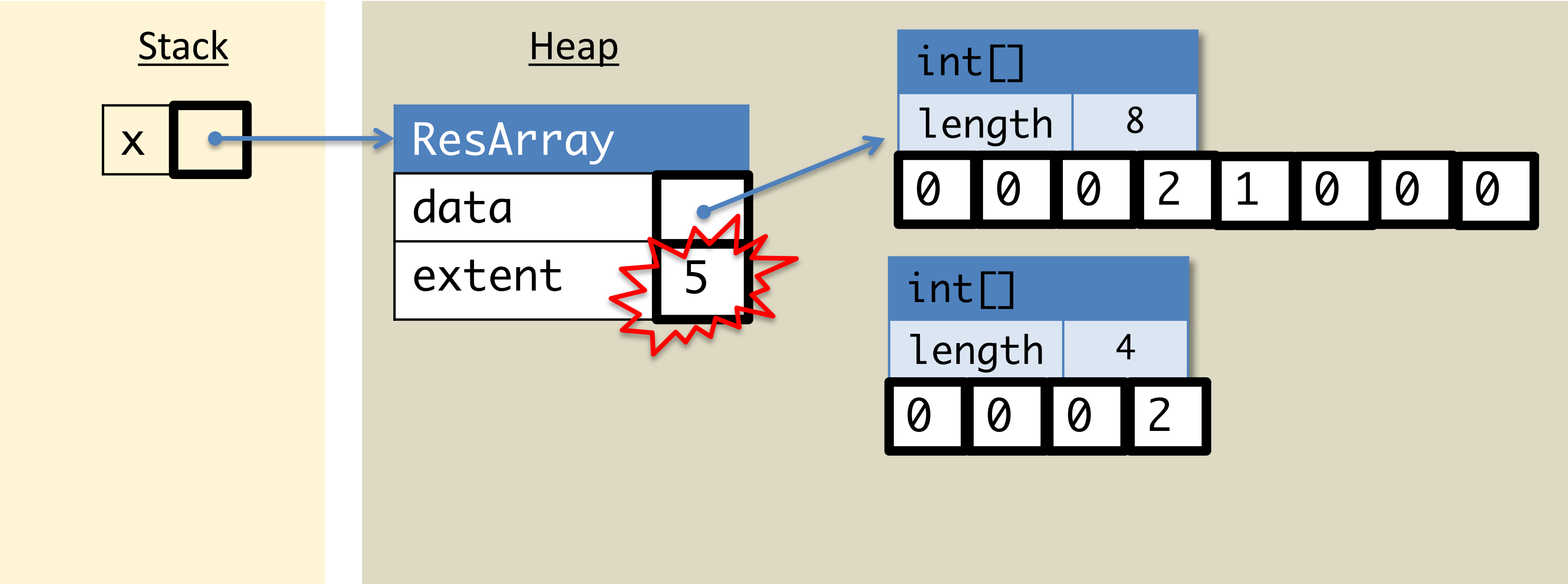
Workspace

~~ResArray x = new ResArray();~~

~~x.set(3,2);~~

~~x.set(4,1);~~

x.set(4,0);



ResArray ASM

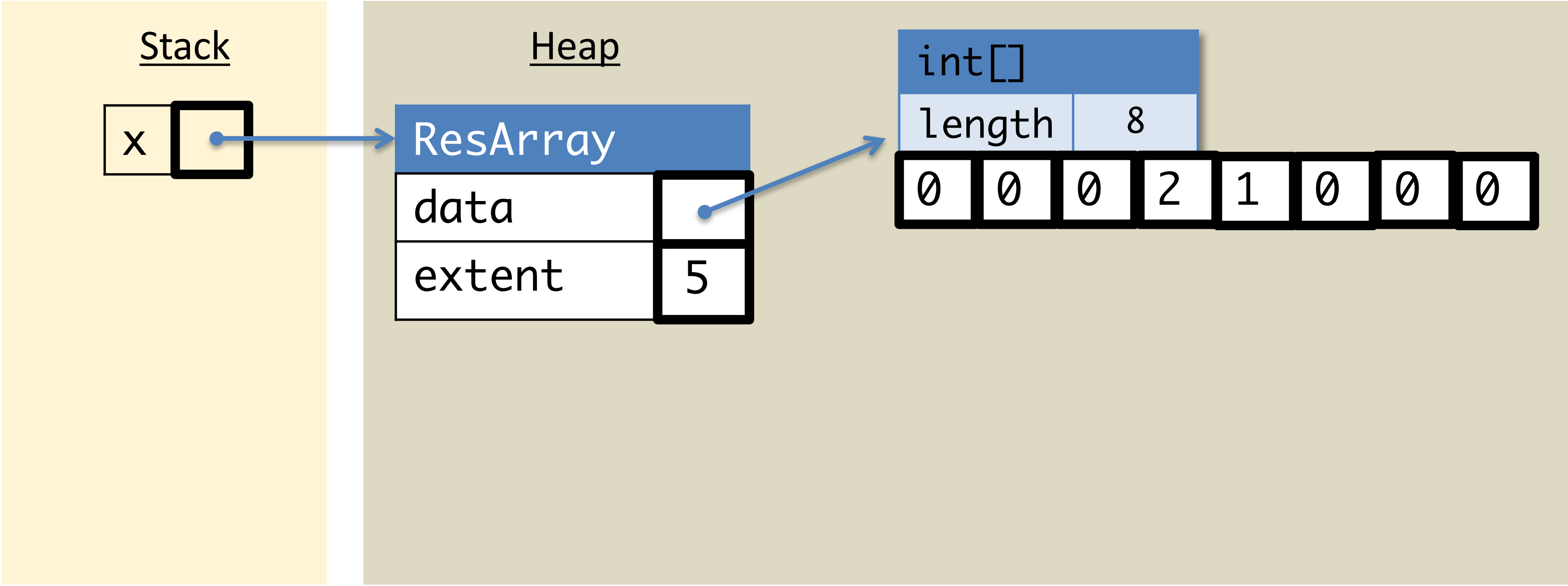
Workspace

~~ResArray x = new ResArray();~~

~~x.set(3,2);~~

~~x.set(4,1);~~

x.set(4,0);



Stack

x

Heap

ResArray

data	
extent	5

int[]

length	8

0	0	0	2	1	0	0	0

ResArray ASM

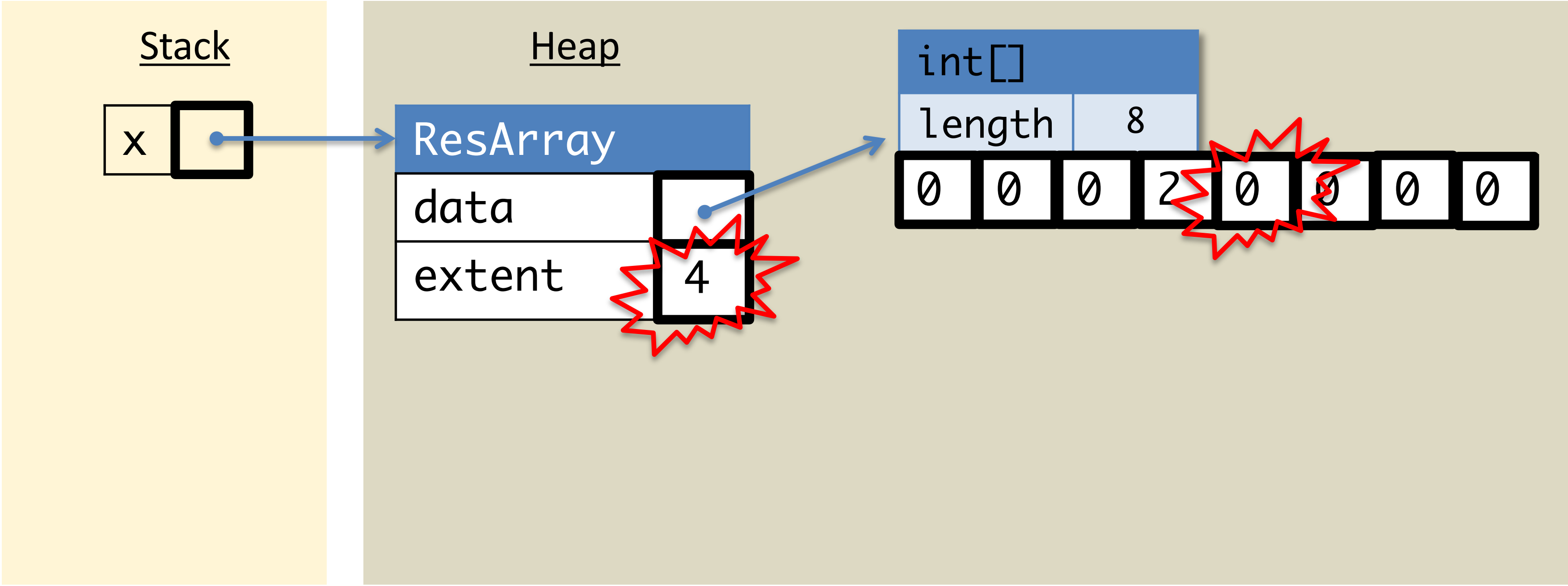
Workspace

~~ResArray x = new ResArray();~~

~~x.set(3,2);~~

~~x.set(4,1);~~

~~x.set(4,0);~~



Stack

Heap

Resizable Arrays

Object Invariant: extent is always 1 past the last nonzero value in data (or 0 if the array is all zeros)

```
public class ResArray {

  /** Constructor, takes no arguments. */
  public ResArray() { … }

  /** Access the array at position i. If position i has not yet
   * been initialized, return 0.
   */
  public int get(int i) { … }

  /** Modify the array at position i to contain the value v. */
  public void set(int i, int v) { … }

  /** Return the extent of the array. */
  public int getExtent() { … }

  /** The smallest prefix of the ResArray
   * that contains all of the nonzero values, as a normal array.
   */
  public int[] values() { … }
}
```

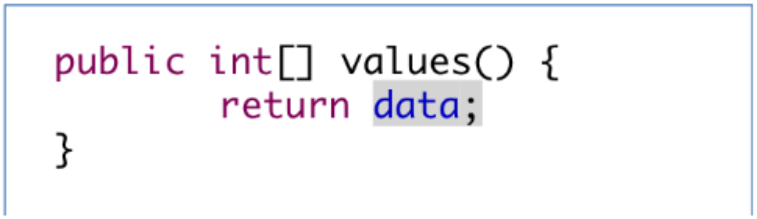
Values Method

```
public int[] values() {
    int[] values = new int[extent];
    for (int i=0; i<extent; i++) {
        values[i] = data[i];
    }
    return values;
}
```

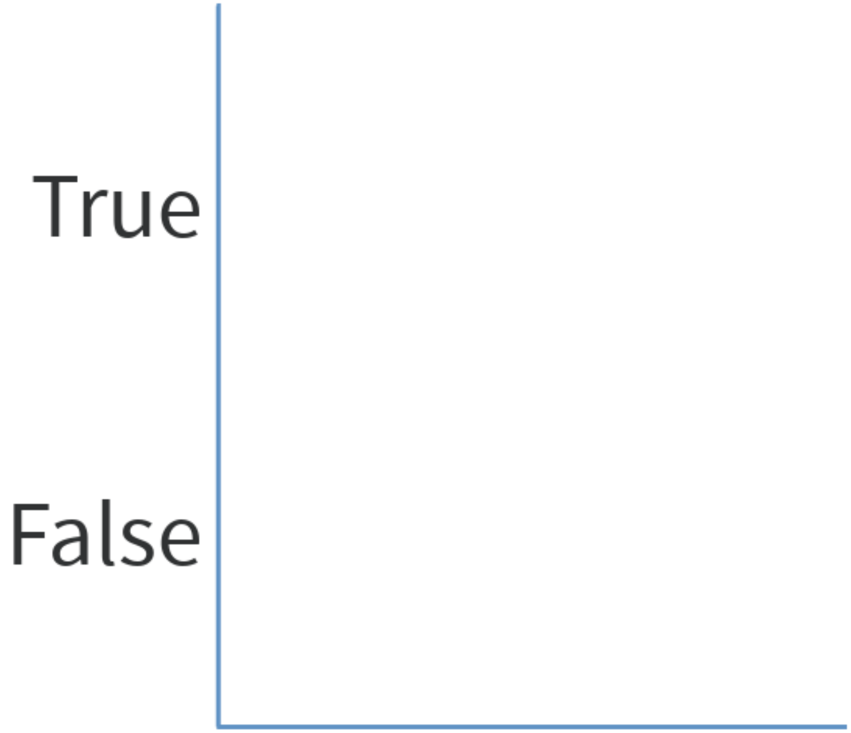
Or maybe we can do it more straightforwardly? …

```
public int[] values() {
    return data;
}
```

This optimized implementation of values correctly encapsulates the state of the ResArray object.



```
public int[] values() {
        return data;
}
```



This optimized implementation of values correctly encapsulates the state of the ResArray object.

```
public int[] values() {
    return data;
}
```

1. True
2. False

Answer: False

ResArray ASM

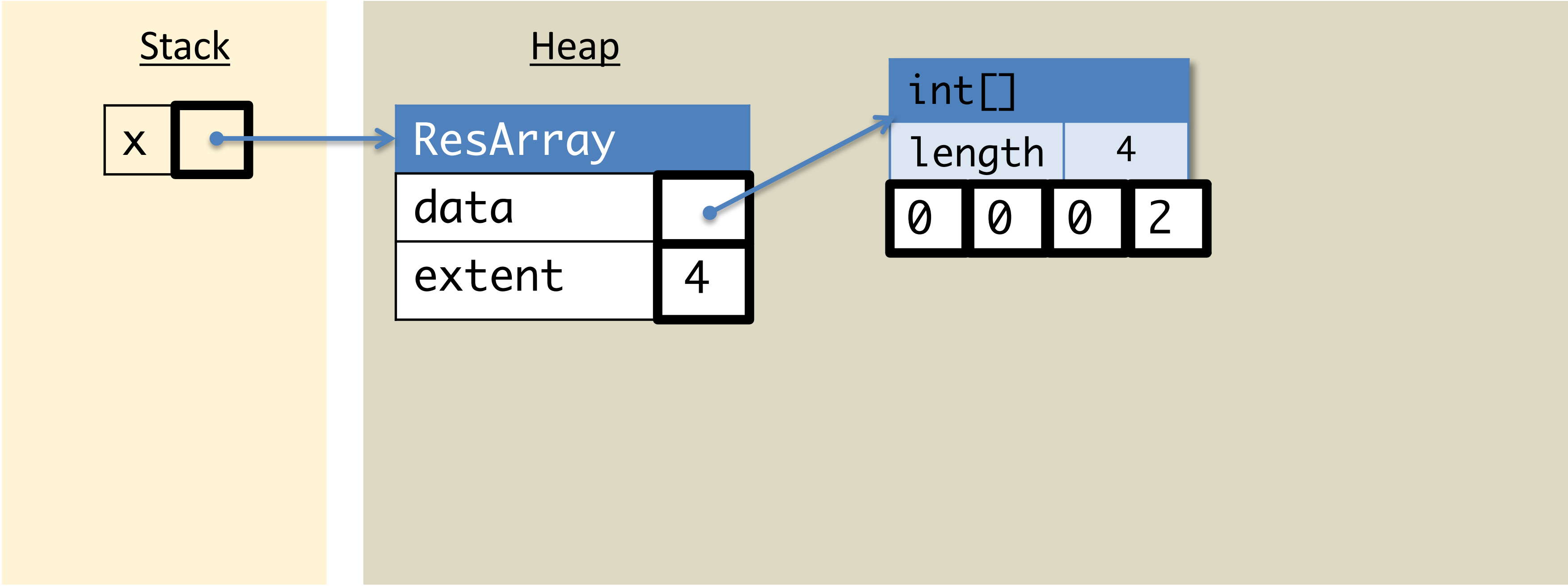
Workspace

~~`ResArray x = new ResArray();`~~

~~`x.set(3,2);`~~

`int[] y = x.values();`

`y[3] = 0;`



ResArray ASM

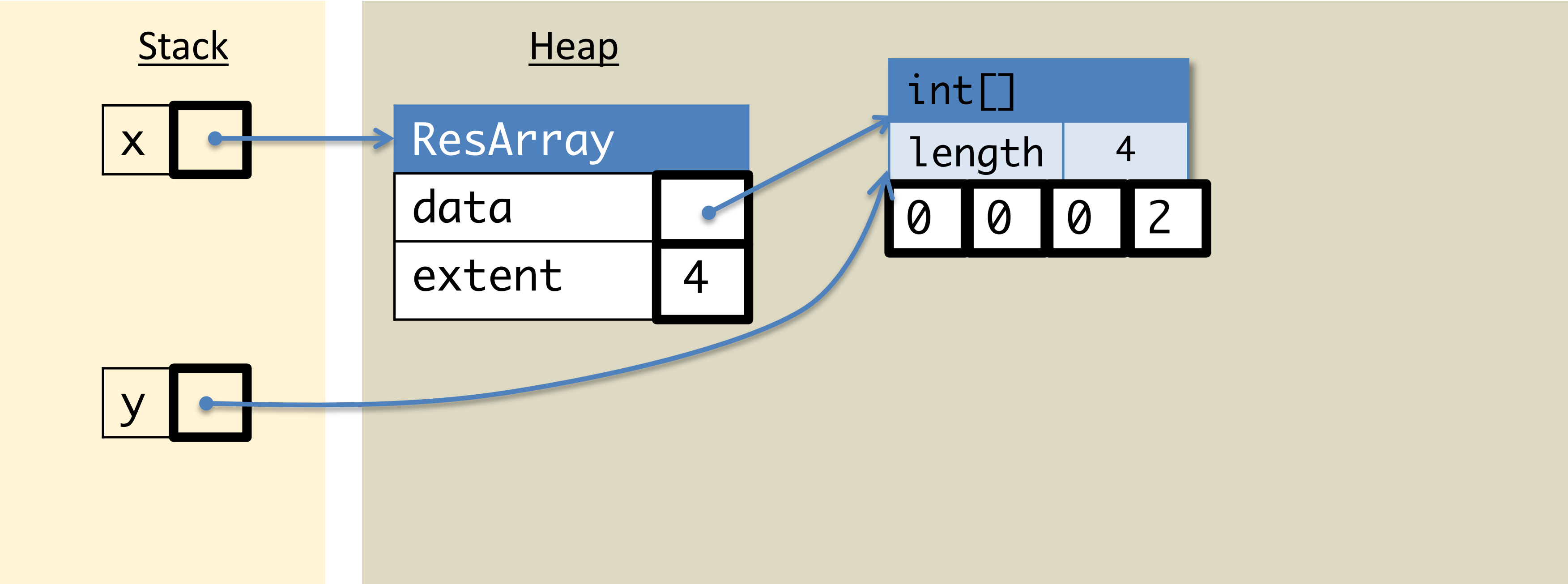
Workspace

~~ResArray x = new ResArray();~~

~~x.set(3,2);~~

~~int[] y = x.values();~~

y[3] = 0;



ResArray ASM

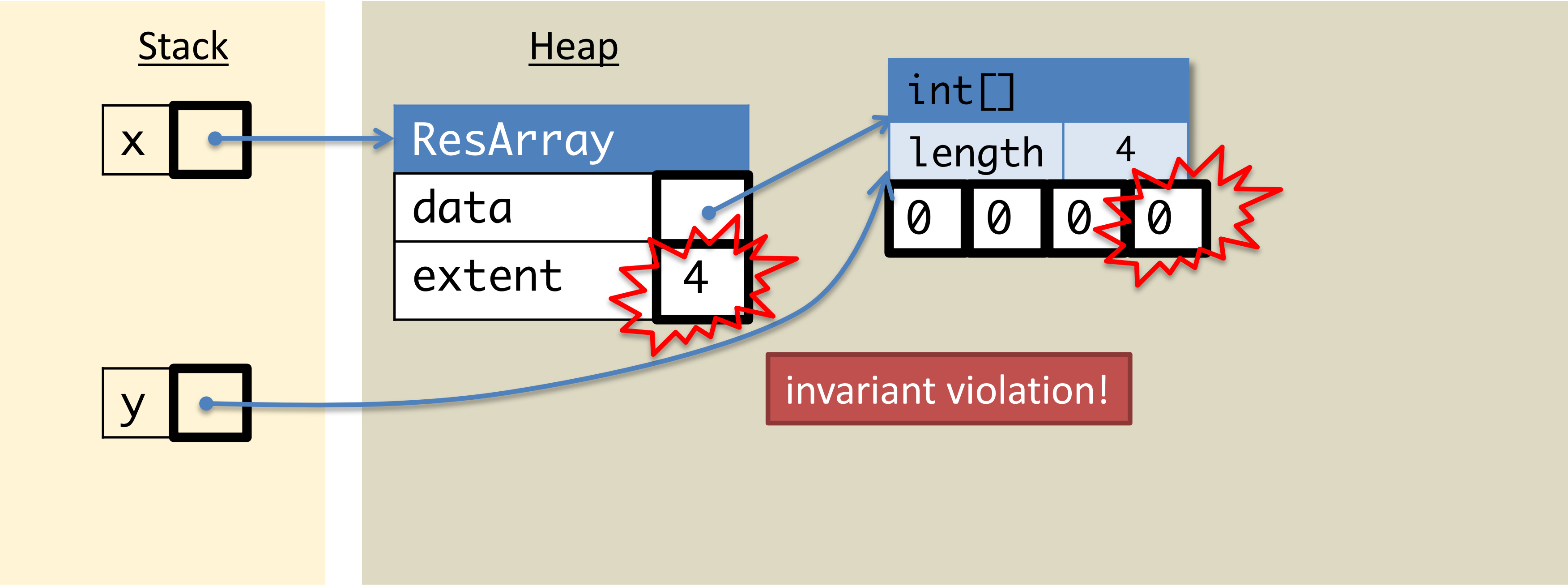
Workspace

~~ResArray x = new ResArray();~~

~~x.set(3,2);~~

~~int[] y = x.values();~~

~~y[3] = 0;~~



invariant violation!

Object encapsulation

- ***All modification to the state of the object must be done using the object's own methods.***

- Use encapsulation to preserve invariants about the state of the object.

- Enforce encapsulation by not returning aliases from methods.