Programming Languages and Techniques (CIS120)

Lecture 24

Resizable Arrays, Java ASM

Chapter 32

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?
- Could use a linked list or queues, but...
  - You want efficient array-indexing operations
- For simplicity: store only integer values*
- Note: Java Standard Library provides ArrayList
  - similar implementation but not the same interface
  - 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 nonzero data in an array */
  public int[] values() { … }

  /** Return the extent of the array.
   *  The extent tells us an upper bound of non-zero data
   *  and is the length of the array returned by values.
   *  @returns 1+index of last nonzero element, or 0 if all
   *  elements are 0.  */
  public int getExtent() { … }
}
```

Abstract Type: ResArray

- `ResArray` class defines a new type with operations
  - `ResArray()` constructor
  - `int get(int i)`
  - `void set(int i, int v)`
  - `int[] values()`
  - `int getExtent()`

- Abstract view: this is an "infinite" array, up to machine limits
  - all nonnegative ints are valid for get and set, no need to keep track of the size* of the array
  - any location that has not been previously set holds the value 0
- Concrete view: we will use a small array to store the nonzero data, and adjust it as necessary

*Java's ArrayList has a notion of "size" or the number of elements stored in the container.

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 ArrayIndexOutOfBounds Exception

The code fails with an IllegalArgumentExce ption

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 thrown
public void testSetNegativeArg () {
    ResArray a = new ResArray();
    Assertions.assertThrows(
       IndexOutOfBoundsException.class, () -> {
             a.set(-17, 23);
       });
}
```

- Which exception to throw? Maybe neither IndexOutOfBoundsException 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

Eclipse Demo

Step 5: Refactor

Can we optimize our implementation?

Refactor 1: How much to resize?

- If the data array is too small, we need to create a larger array and copy the new data over
- How big should the new array be?
  - Must have length at least new index + 1 so that we can store value
  - May be larger, so that we don't need to make so many copies
  - Don't want to be too large, so that we don't waste space in the heap

Big idea: at least double* the array each time

- If we insert values in order, how many times will each value be copied as the array grows?

*Many libraries, including ArrayList, scale up by 1.5 instead of 2. But the idea is the same.

Resizing Factor: index+1

If we insert values in order how many copies?

```
a.set(0, ..); // grow to 1
a.set(1, ..); // grow to 2,  copy 0
a.set(2, ..); // grow to 3,  copy 0,1
a.set(3, ..); // grow to 4,  copy 0,1,2,
a.set(4, ..); // grow to 5,  copy 0,1,2,3
a.set(5, ..); // grow to 6,  copy 0,1,2,3,4
a.set(6, ..); // grow to 7,  copy 0,1,2,3,4,5
a.set(7, ..); // grow to 8,  copy 0,1,2,3,4,5,6
a.set(8, ..); // grow to 9,  copy 0,1,2,3,4,5,6,7
```

Total number of copies: 8+7+6+5+4+3+2+1 = 36

no wasted space

Resizing Factor: doubling

If we insert values in order how many copies?

```
a.set(0, ..); // grow to 1
a.set(1, ..); // grow to 2,  copy 0
a.set(2, ..); // grow to 4,  copy 0,1
a.set(3, ..);
a.set(4, ..); // grow to 8,  copy 0,1,2,3
a.set(5, ..);
a.set(6, ..);
a.set(7, ..);
a.set(8, ..); // grow to 16, copy 0,1,2,3,4,5,6,7
```

Total number of copies: 8+4+2+1 = 15

- as the array grows larger, difference is even more significant (see CIS 121)
- trade-off: wastes at most 2x space

Refactor 2: extent

- Add a new private instance variable

```
private int extent = 0;
```

- Modify getExtent to return this value instead of recalculating

```
public int getExtent() {
  return extent;
}
```

- What invariant should we maintain about this state?

INVARIANT: extent = 1 + index of last nonzero element, or 0 if all elements are 0

Update *extent* when data changes

```
/* 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--;
      }
    }
  }
```

Refactor 3: Optimize values(?)

```
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 this way? …

```
public int[] values() {
    if (data.length == extent) {
      return data;
    } else {
      // as above
    }
}
```

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.

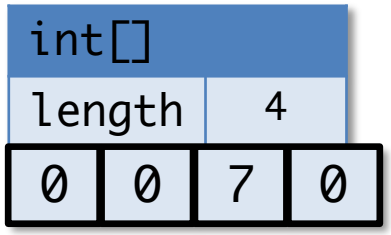
Type	Description	Values
`byte`	8-bit	`-128 to 127`
`short`	16-bit integer	`-32768 to 32767`
`int`	32-bit integer	-2^{31} to $2^{31} - 1$
`long`	64-bit integer	-2^{63} to $2^{63} - 1$
`float`	32-bit IEEE floating point	
`double`	64-bit IEEE floating point	
`boolean`	true or false	`true false`
`char`	16-bit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

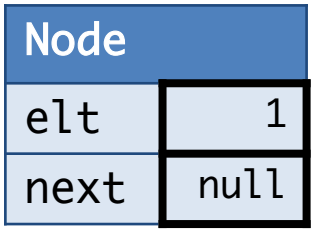


length *never* mutable; elements *always* mutable

Objects

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

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



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);
```

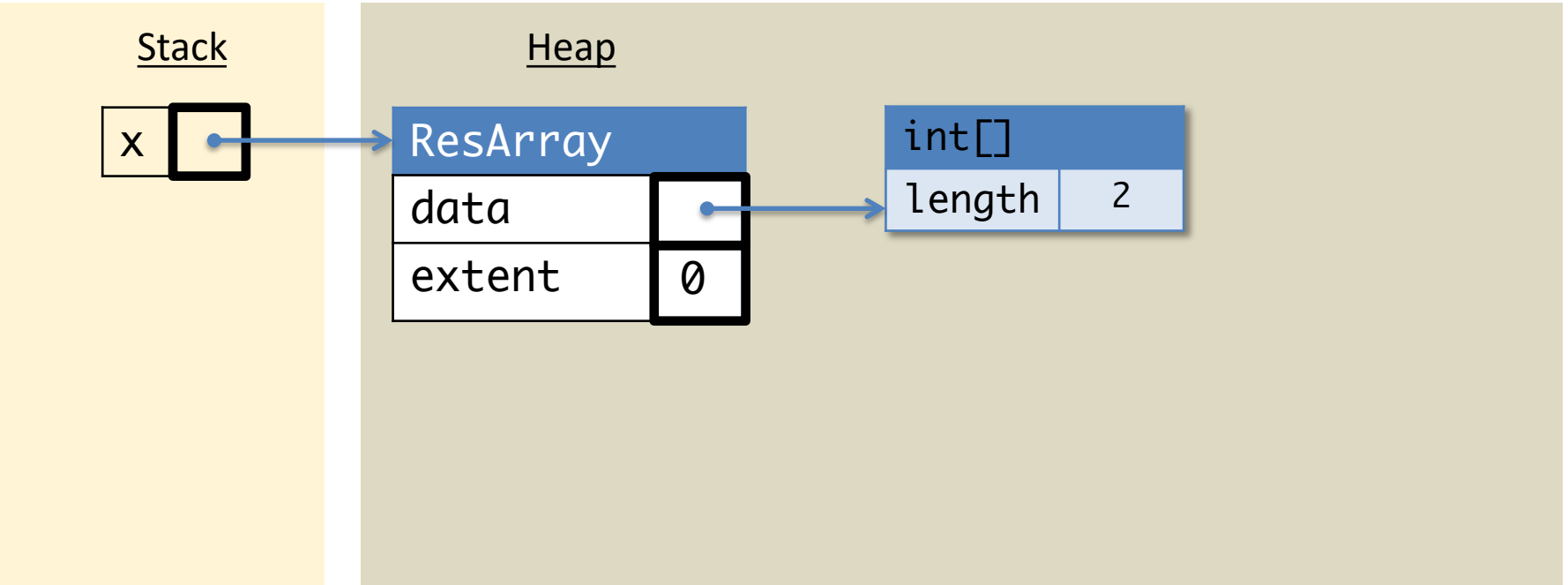
Stack

Heap

ResArray ASM

Workspace

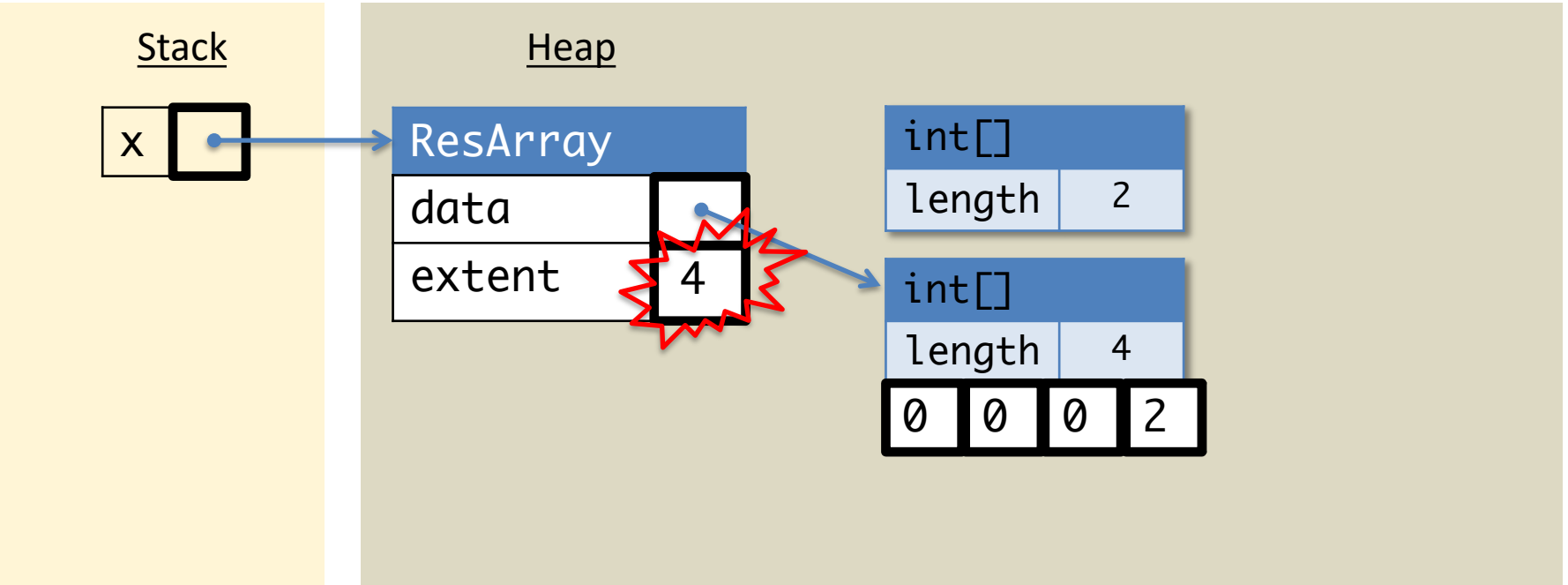
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

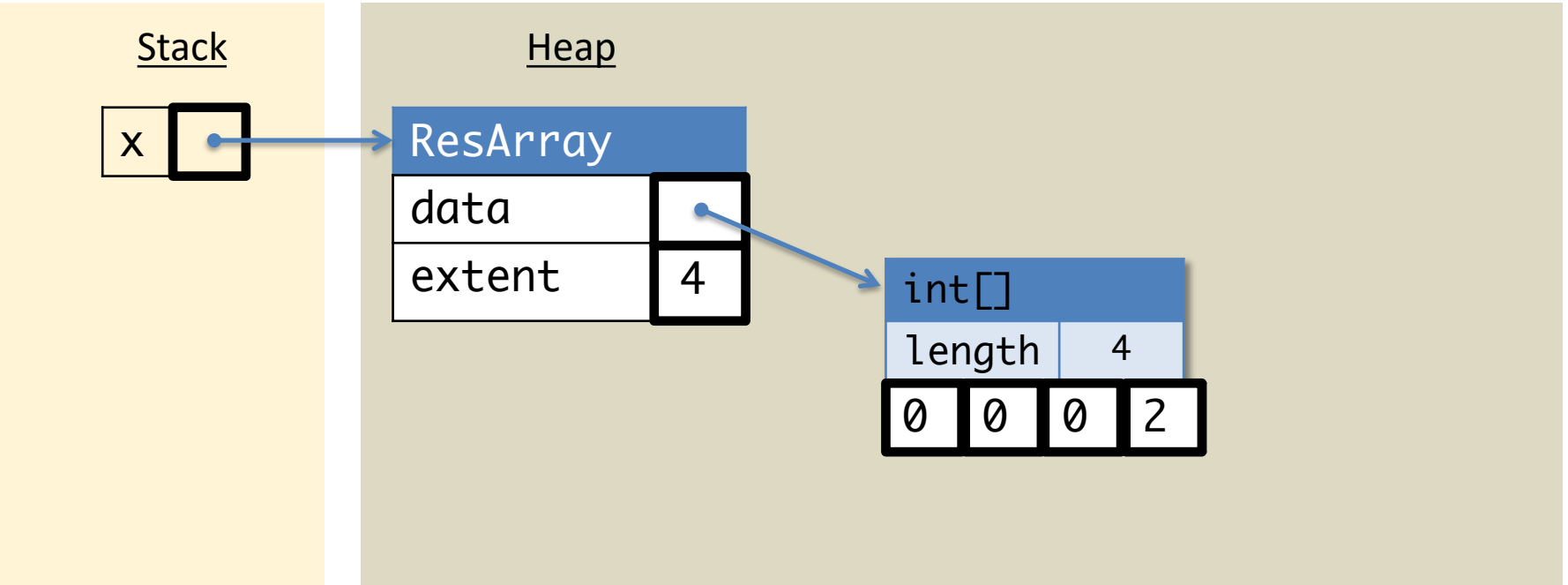
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

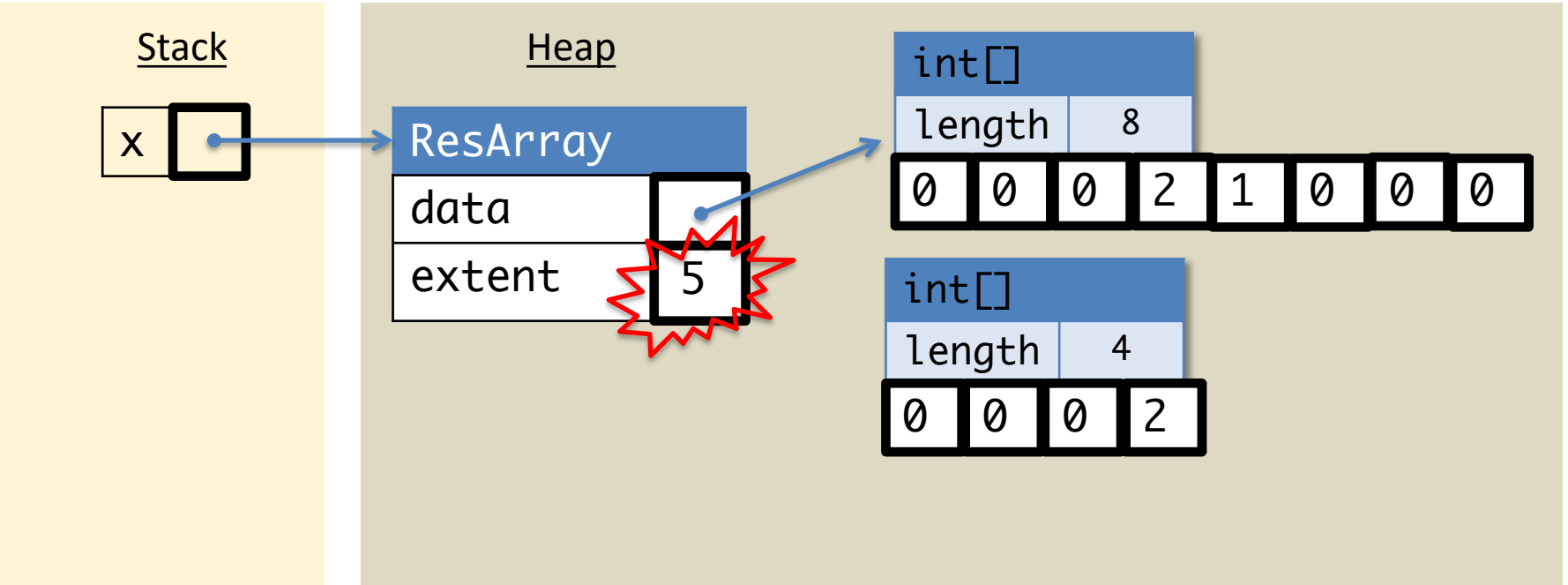
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

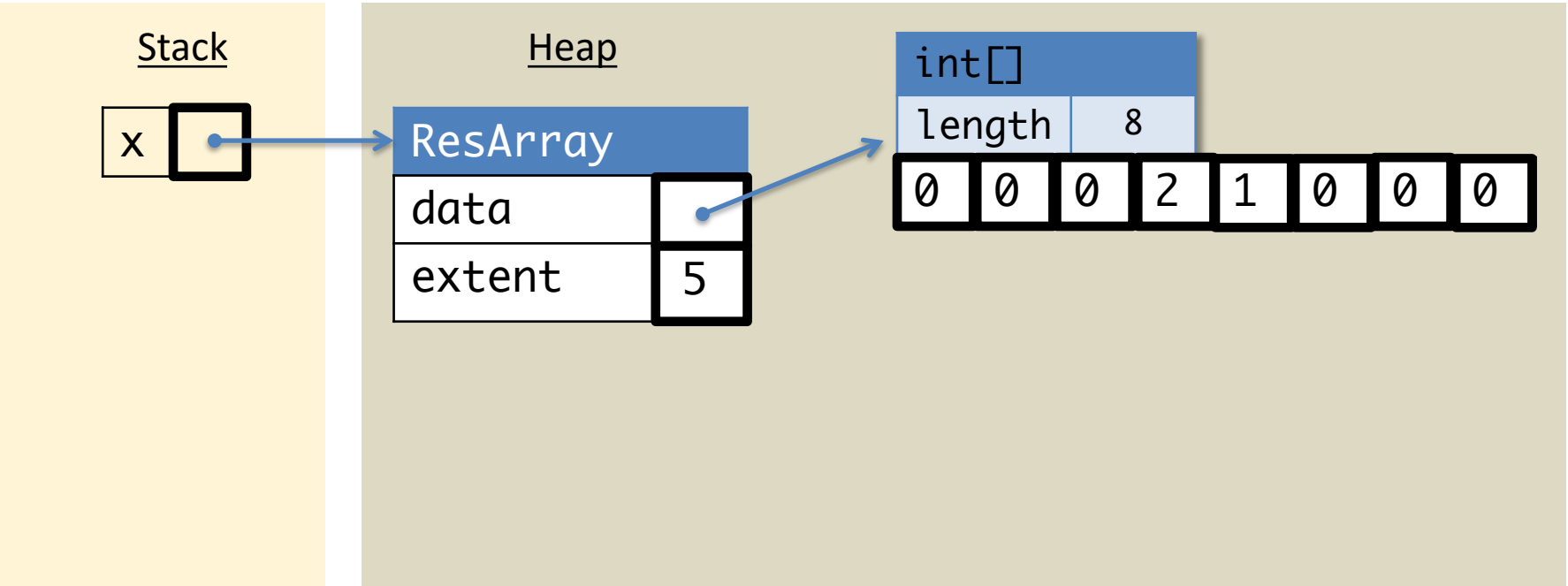
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

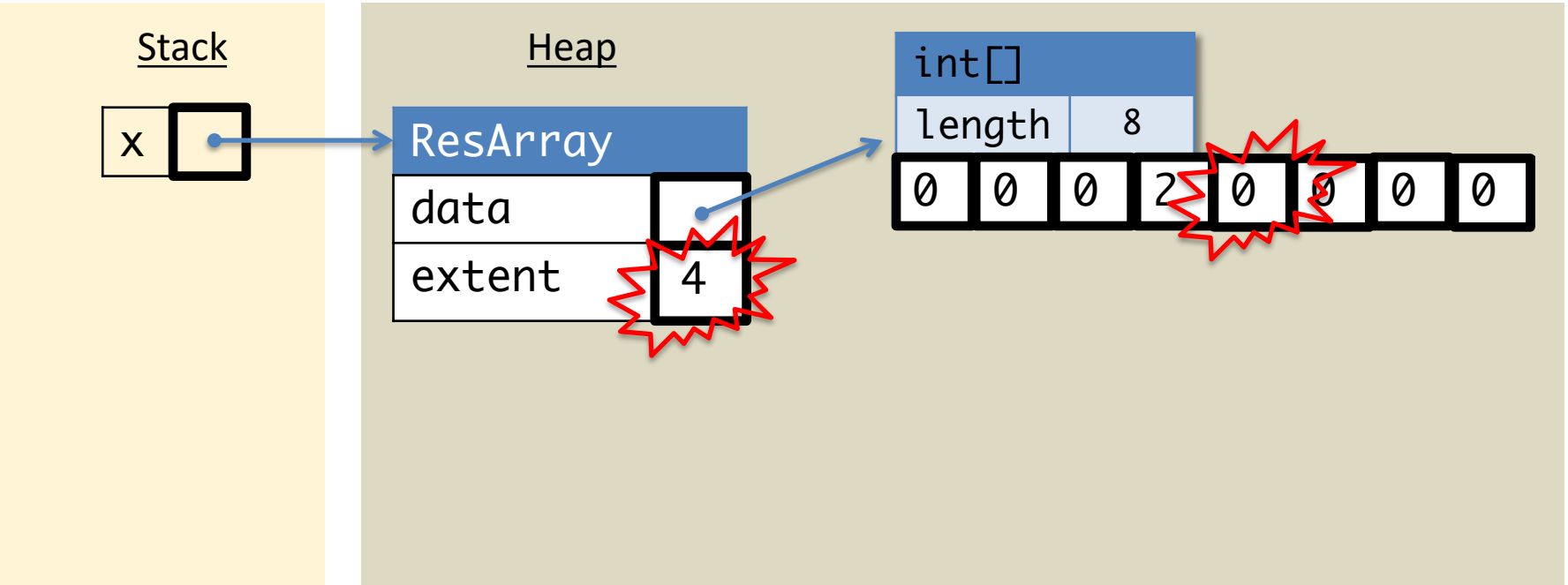
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

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



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() { … }
}
```

Refactor 3: Optimize values

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

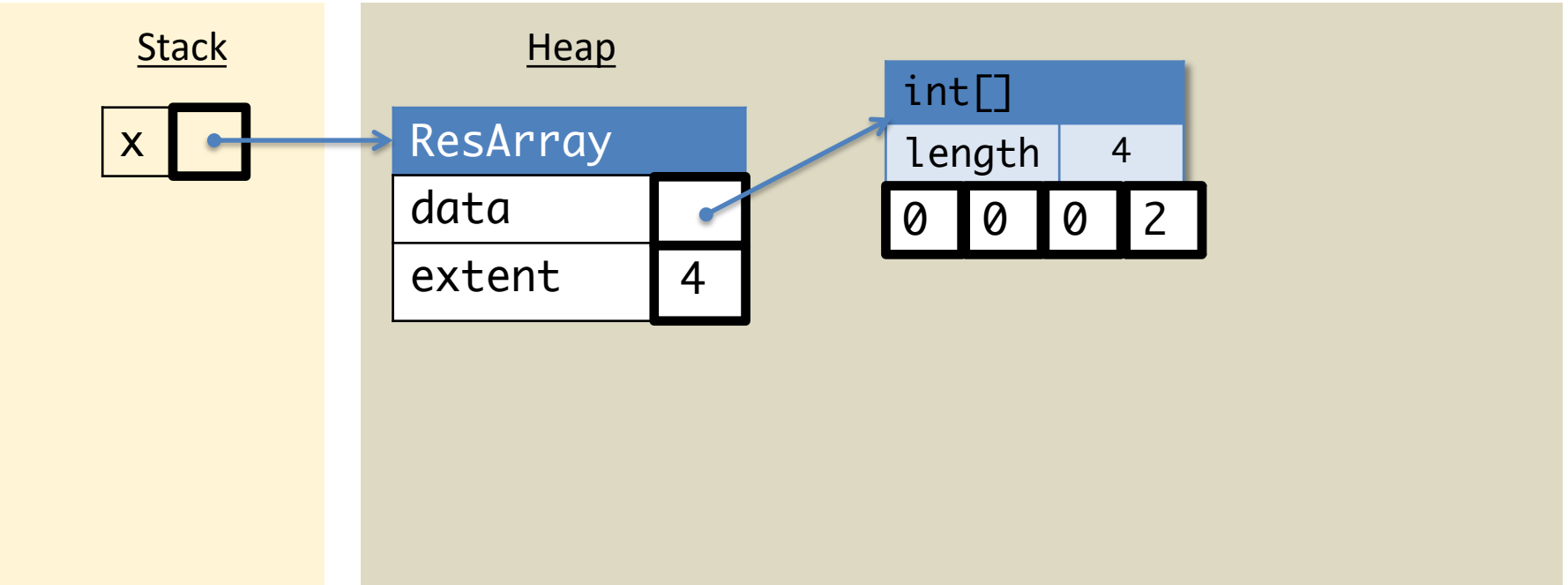
Or maybe we can do it this way? …

```
public int[] values() {
    if (data.length == extent) {
      return data;
    } else {
      // as above
    }
}
```

ResArray ASM

Workspace

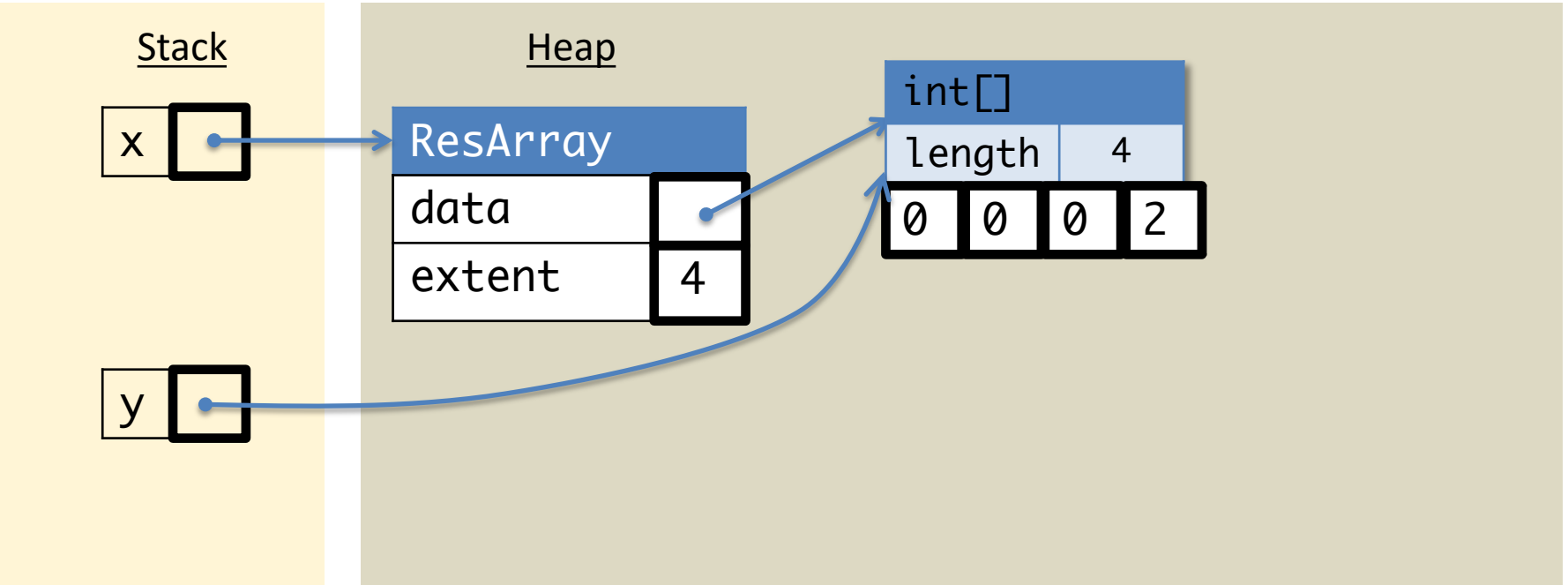
```
ResArray x = new ResArray();
x.set(3,2);
int[] y = x.values();
y[3] = 0;
```



ResArray ASM

Workspace

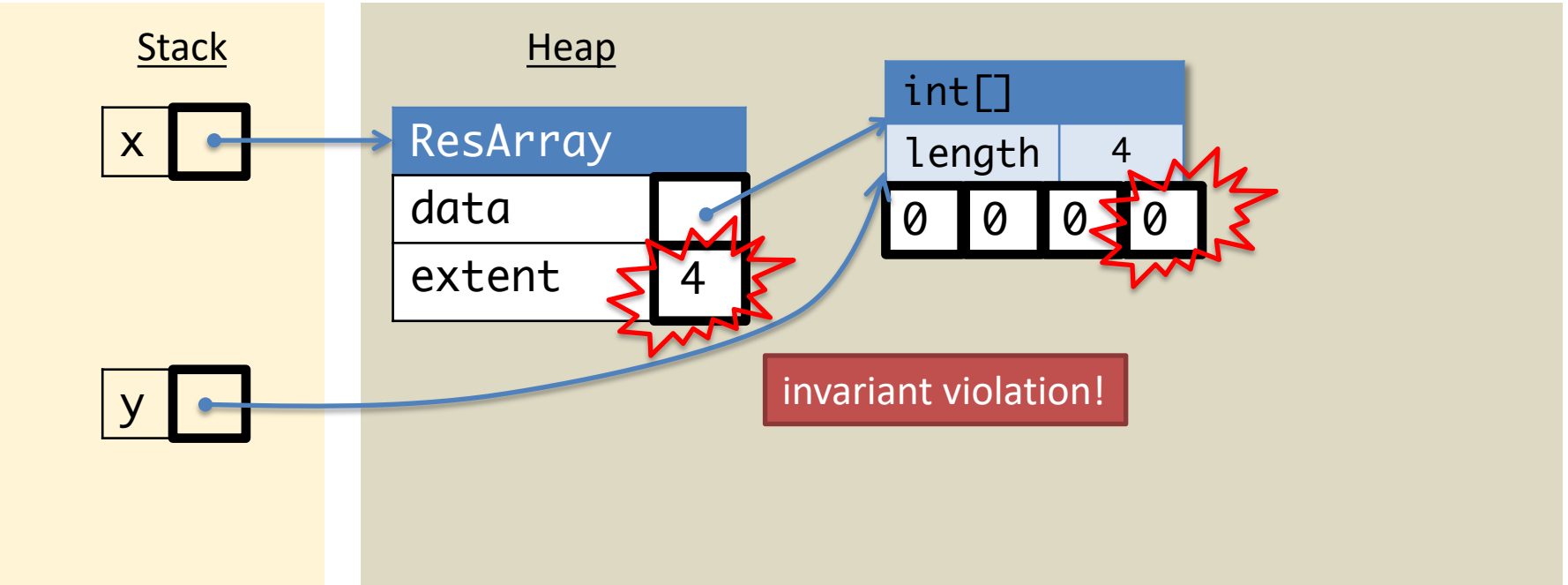
```
ResArray x = new ResArray();
x.set(3,2);
int[] y = x.values();
y[3] = 0;
```



ResArray ASM

Workspace

```
ResArray x = new ResArray();
x.set(3,2);
int[] y = x.values();
y[3] = 0;
```



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.