

Programming Languages and Techniques (CIS120)

Lecture 37

April 22, 2016

Encapsulation & Hashing

How is the Game Project going so far?

1. not started
2. got an idea
3. submitted design proposal
4. started coding
5. it's somewhat working
6. it's mostly working
7. debugging / polishing
8. done!

Announcements

- Monday is the bonus lecture! No clickers required.
"Code is Data"
- Game project due Tuesday at midnight, hard deadline.
NO LATE PROJECTS WILL BE ACCEPTED.

FINAL EXAM

- **Monday, May 9th, 9-11AM**
 - **Four locations, see webpage for details**
- *Comprehensive* exam over course concepts:
 - OCaml material (though we won't worry much about syntax)
 - All Java material (emphasizing material since midterm 2)
 - all course content
 - old exams posted
- Closed book, but:
 - One letter-sized, *handwritten* sheet of notes allowed
- Review Session:
 - TBA

```
public class ResArray {
    /** Constructor, takes no arguments. */
    public ResArray() { ... }
    /** Access position i. If position i has not yet
     * been initialized, return 0. */
    public int get(int idx) { ... }
    /** Update index i to contain the value v. */
    public void set(int idx, int val) { ... }
    /** Return the extent of the array. i.e.
     one past the index of the last nonzero value in the array. */
    public int getExtent() { ... }
}
```

```
ResArray a = new ResArray();
a.set(3,2);
a.set(4,1);
a.set(4,0);
int result = a.getExtent();
```

What should be the result?

1. 0
2. 3
3. 4
4. 5
5. ArrayIndexOutOfBoundsException
6. NullPointerException

Resizable Arrays

```
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() { ... }  
  
}
```

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

ResArray ASM

Workspace

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

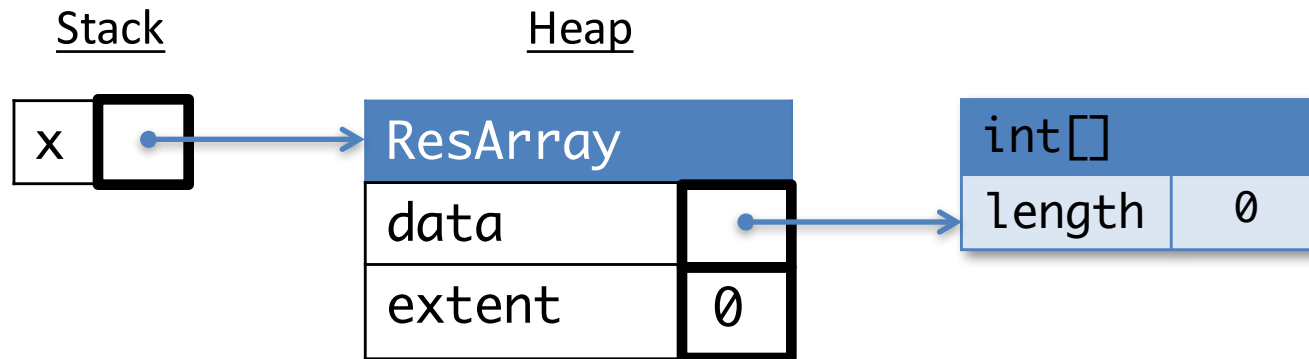
Stack

Heap

ResArray ASM

Workspace

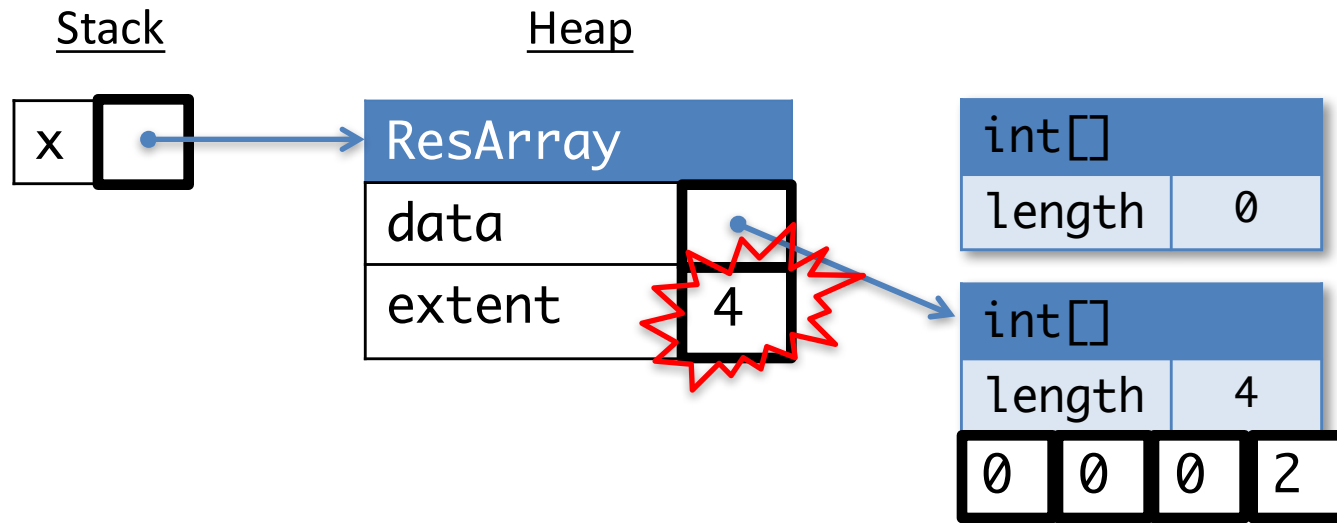
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM

Workspace

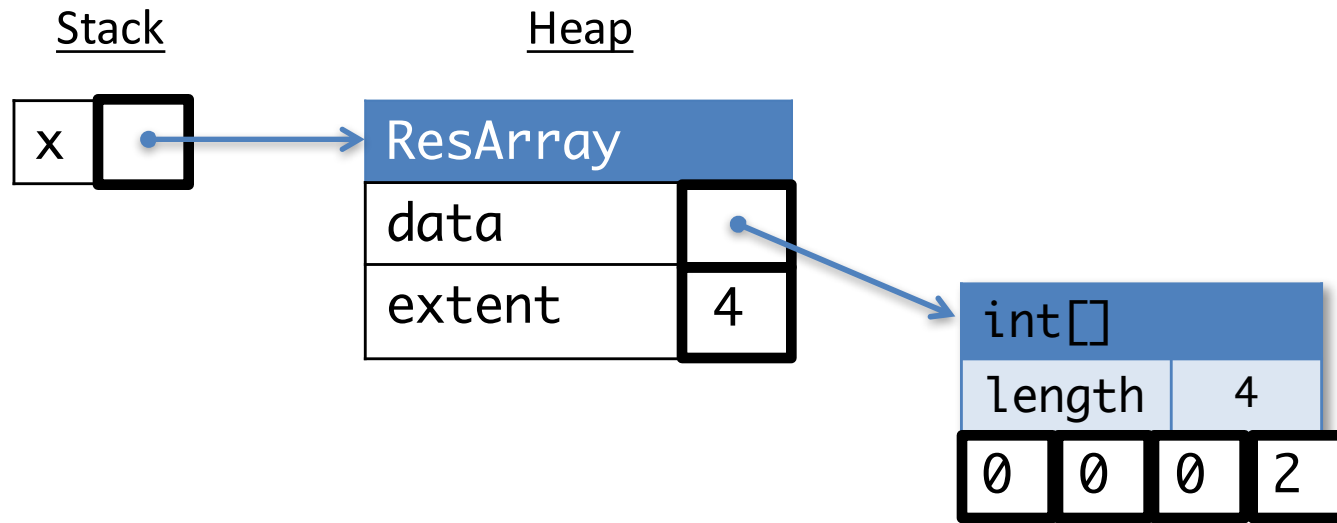
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM

Workspace

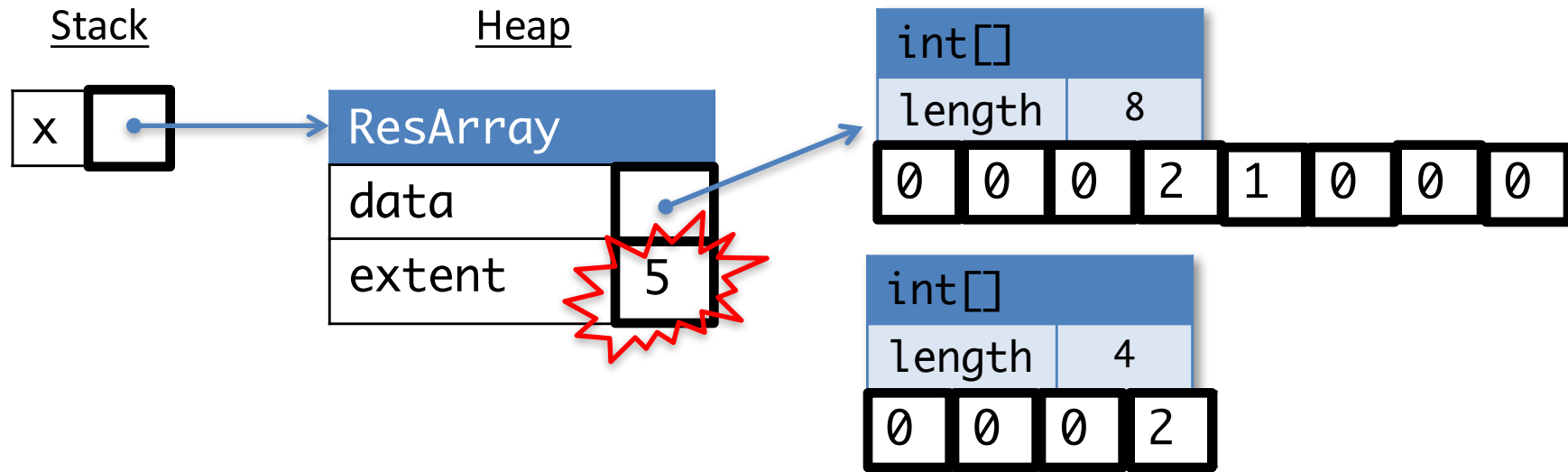
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM

Workspace

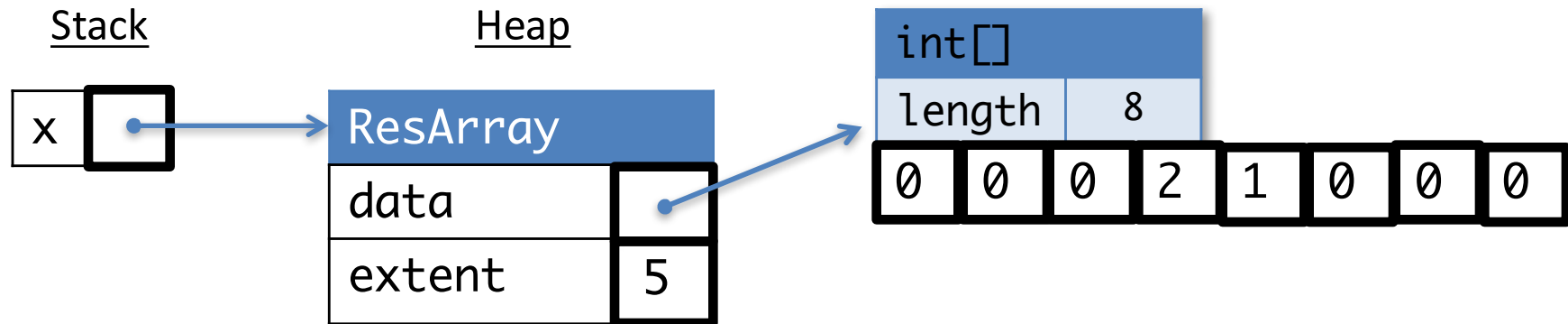
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM

Workspace

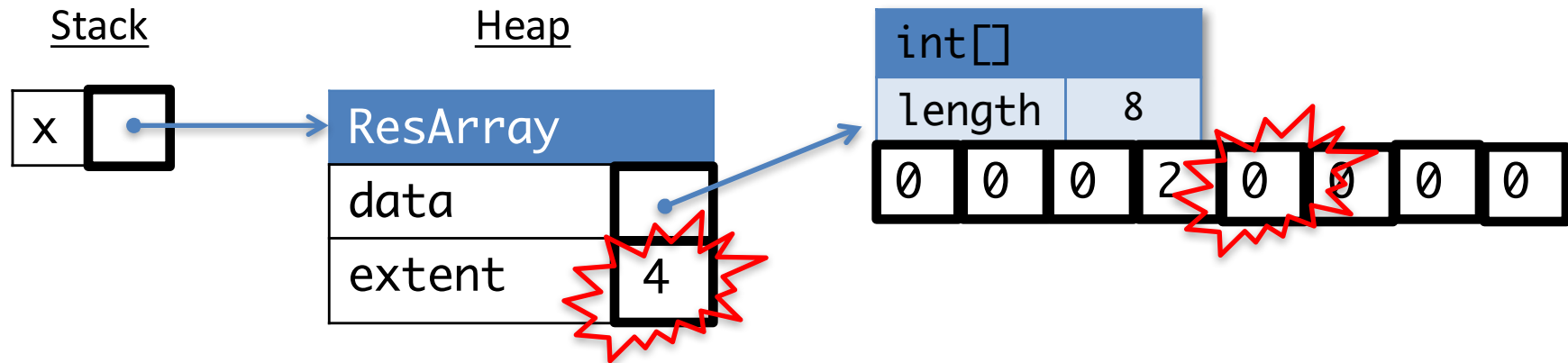
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM


Workspace

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



Resizable Arrays

```
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() { ... }  
  
    /** An array containing all nonzero values */  
    public int[] values() { ... }  
}
```



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

Values Method

```
public class ResArray {  
  
    private int[] data;  
    private int extent = 0;  
  
    ...  
  
    /** An array containing all nonzero values */  
    public int[] values() {  
        return data;  
    }  
  
}
```

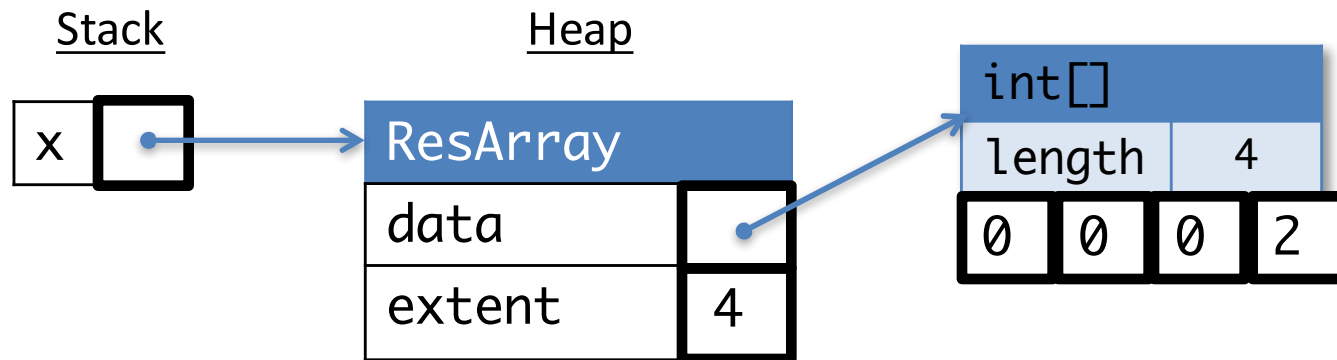
Object Invariant: extent is
always 1 past the last nonzero
value in data
(or 0 if the array is all zeros)

Is this a good implementation of
"values" ?

ResArray ASM

Workspace

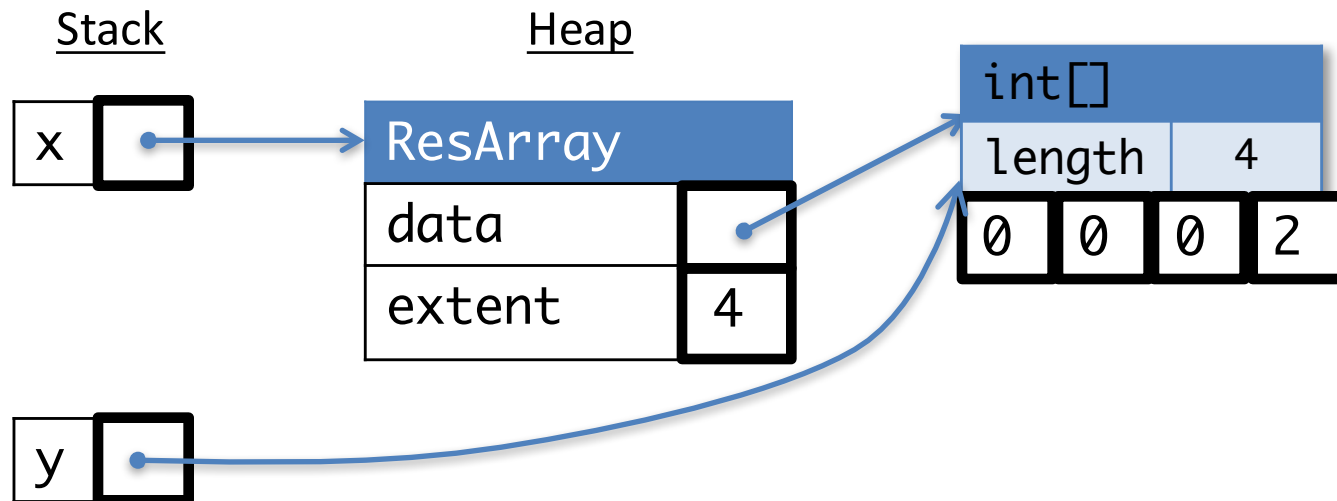
```
ResArray x = new ResArray();  
x.set(3,2);  
int[] y = x.values();  
y[3] = 0;
```



ResArray ASM

Workspace

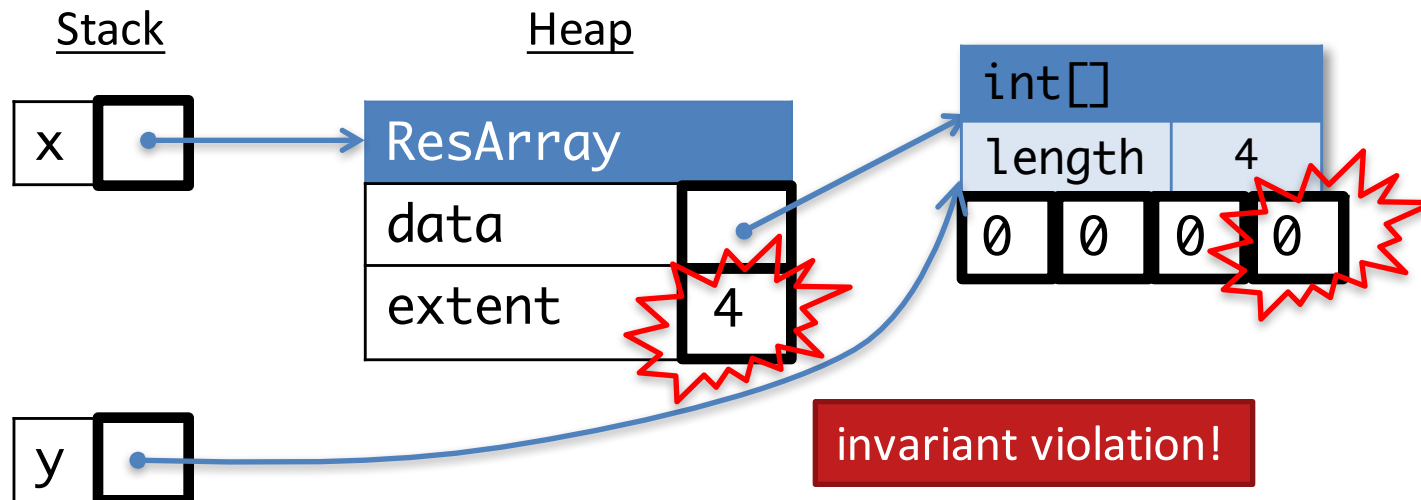
```
ResArray x = new ResArray();  
x.set(3,2);  
int[] y = x.values();  
y[3] = 0;
```



ResArray ASM

Workspace

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



Values method and encapsulation

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

- Use encapsulation to preserve invariants about the state of the object.
- All modification to the state of the object must be done using the object's own methods.
- Enforce encapsulation by not returning aliases to mutable data structures from methods.

Hash Sets & Hash Maps

array-based implementation of sets and maps

Hash Sets and Maps: The Big Idea

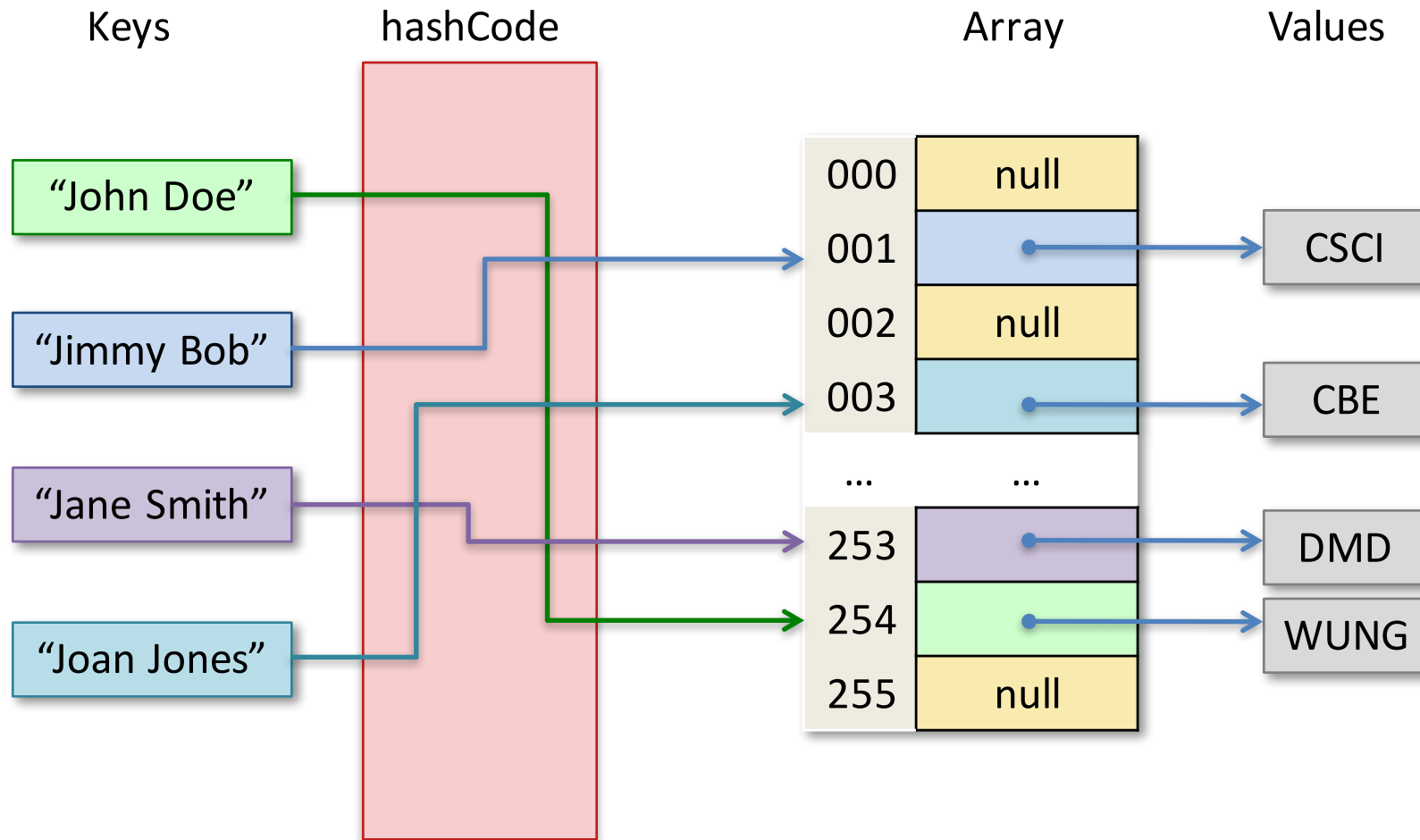
Combine:

- the advantage of arrays:
 - *efficient* random access to its elements
- with the advantage of a map datastructure
 - arbitrary keys (not just integer indices)

How?

- Create an index into an array by *hashing* the data in the key to turn it into an int
 - Java's hashCode method maps key data to ints
 - Generally, the space of keys is much larger than the space of hashes, so, unlike array indices, hashCodes might not be unique

Hash Maps, Pictorially

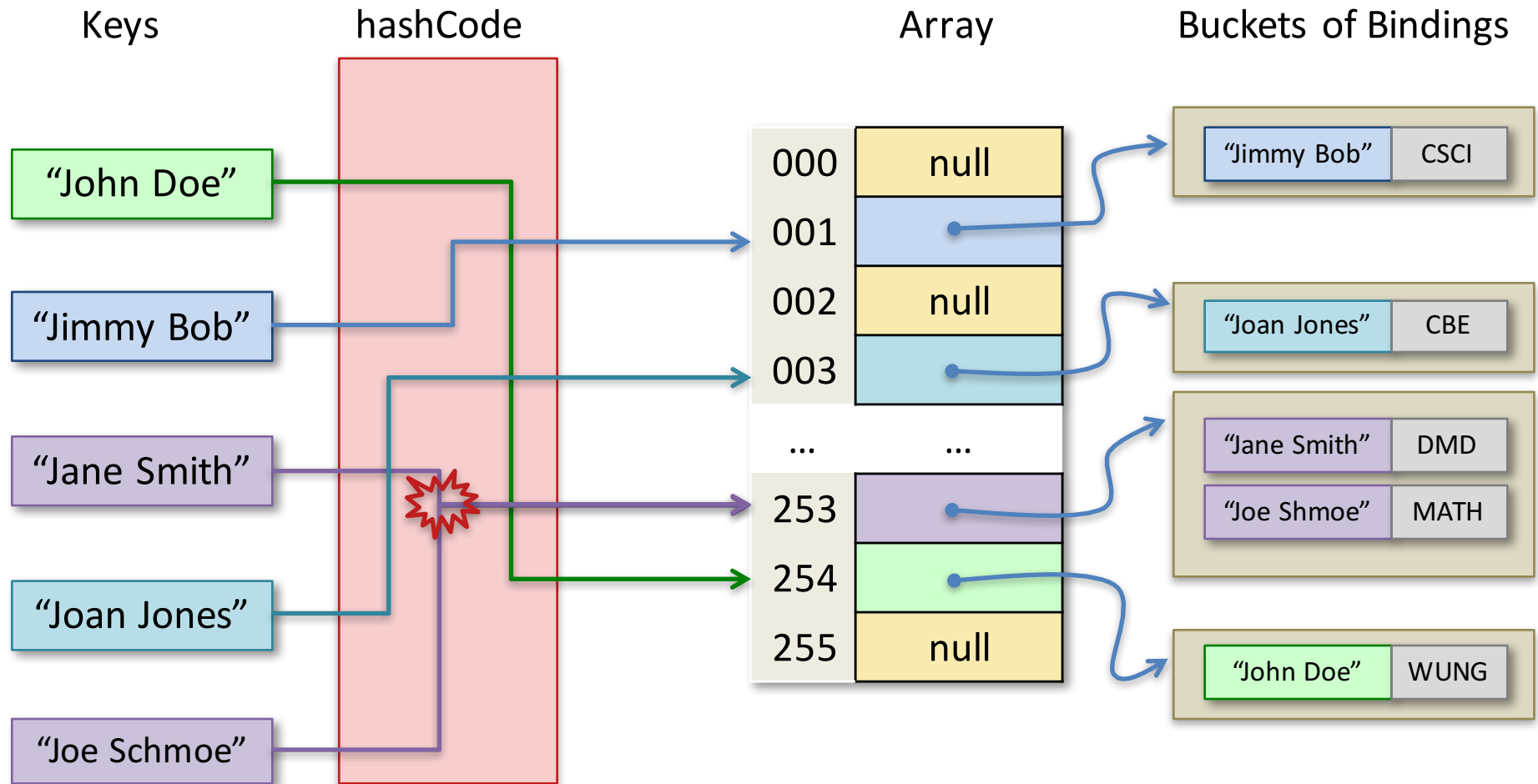


A schematic HashMap taking Strings (student names) to Undergraduate Majors. Here, "John Doe".hashCode() returns an integer n , its *hash*, such that $n \bmod 256$ is 254.

Hash Collisions

- Uh Oh: Indices derived via hashing may not be unique!
“Jane Smith”.hashCode() % 256 → 253
“Joe Schmoe”.hashCode() % 256 → 253
- Good hashCode functions make it *unlikely* that two keys will produce the same hash
- But, it can happen that two keys do produce the same index – that is, their hashes *collide*

Bucketing and Collisions



Here, "Jane Smith".hashCode() and "Joe Schmoe".hashCode() happen to collide. The bucket at the corresponding index of the Hash Map array stores the map data.

Bucketing and Collisions

- Using an array of *buckets*
 - Each bucket stores the mappings for keys that have the same hash.
 - Each bucket is itself a map from keys to values (implemented by a linked list or binary search tree).
 - The buckets can't use hashing to index the values – instead they use key equality (via the key's equals method)
- To lookup a key in the Hash Map:
 - First, find the right bucket by indexing the array through the key's hash
 - Second, search through the bucket to find the value associated with the key
- Not the only solution to the collision problem

Hashing and User-defined Classes

```
public class Point {  
    private final int x;  
    private final int y;  
    public Point(int x, int y) { this.x = x; this.y = y;  
}  
  
    public int getX() { return x; }  
    public int getY() { return y; }  
}
```

// somewhere in main...

```
Map<Point,String> m = new HashMap<Point,String>();  
m.put(new Point(1,2), "House");  
System.out.println(m.containsKey(new Point(1,2)));
```

What gets printed to the console?

1. true
2. false
3. I have no idea

HashCode Requirements

Whenever you override `equals` you must also override `hashCode` in a consistent way:

- whenever `o1.equals(o2) == true` you must ensure that `o1.hashCode() == o2.hashCode()`

Why? Because comparing hashes is supposed to be a quick approximation for equality.

- Note: the converse does not have to hold:
 - `o1.hashCode() == o2.hashCode()` does *not* necessarily mean that `o1.equals(o2)`

Example for Point

```
public class Point {  
    @Override  
    public int hashCode() {  
        final int prime = 31;  
        int result = 1;  
        result = prime * result + x;  
        result = prime * result + y;  
        return result;  
    }  
}
```

- Examples:
 - (new Point(1,2)).hashCode() yields 994
 - (new Point(2,1)).hashCode() yields 1024
- Note that *equal* points have the same hashCode
- Why 31? Prime chosen to create more uniform distribution
- Note: eclipse can generate this code

Computing Hashes

- What is a good recipe for computing hash values for your own classes?
 - intuition: “smear” the data throughout all the bits of the resulting integer
- 1. Start with some constant, arbitrary, non-zero int in `result`.
- 2. For each significant field `f` of the class (i.e. each field taken into account when computing equals), compute a “sub” hash code `C` for the field:
 - For boolean fields: $(f ? 1 : 0)$
 - For byte, char, int, short: $(int) f$
 - For long: $(int) (f ^ (f >>> 32))$
 - For references: 0 if the reference is null, otherwise use the `hashCode()` of the field.
- 3. Accumulate those subhashes into the result by doing (for each field’s `C`):
`result = prime * result + c;`
- 4. return `result`

Hash Map Performance

- Hash Maps can be used to efficiently implement Maps and Sets
 - There are many different strategies for dealing with hash collisions with various time/space tradeoffs
 - Real implementations also dynamically rescale the size of the array (which might require re-computing the bucket contents)
- If the hashCode function gives a good (close to uniform) distribution of hashes the buckets are expected to be small (only one or two elements)
- Performance depends on workload

Terminological Clash

- The word "hash" is also used in cryptography
- SHA-1, SHA-2, SHA-3, MD5, etc.
- Cryptographic hashes are intended to reduce large byte sequences to short byte sequences
 - Very hard to invert
 - Should only rarely have collisions
 - Are considerably more expensive to compute than hashCode (so not suitable for hash tables)
- Never use hashCode when you need a cryptographic hash!
 - See CIS 331 for more details

Collections: take away lessons

equals

hashCode

compareTo

Collections Requirements

- All collections use `equals`
 - Defaults to `==` (reference equality)
 - Override `equals` to create structural equality
 - Should be: false for distinct instance classes
 - An equivalence relation: reflexive, symmetric, transitive
- HashSets/HashMaps use `hashCode`
 - Override when `equals` is overridden
 - Should be compatible with `equals`
 - Should try to "distribute" the values uniformly
 - Iterator not guaranteed to follow element order
- Ordered collections (`TreeSet`, `TreeMap`) need to implement `Comparable<Object>`
 - Override `compareTo`
 - Should implement a *total order*
 - Strongly recommended to be compatible with `equals`
(i.e. `o1.equals(o2)` exactly when `o1.compareTo(o2) == 0`)

Comparing Collection Performance

HashTest.java