CSE373: Data Structures & Algorithms

Software-Design Interlude – Preserving Abstractions

Riley Porter

Winter 2017

Course Logistics

- HW1 extra credit concerns: I know how Extra Credit works. Canvas doesn't.
- HW2 due. Last day to submit with late days is tonight.
- HW3 out, topic is hashing. See slides from last Wednesday and Friday (same slide deck)

Today's Topic: Abstractions

- The ADTs we cover in class are important to know conceptually but "in real life", they'll be provided by libraries
- The key idea of an ***abstraction*** arises *all the time*
  - Clients do not know how it is implemented
  - Clients do not need to know
  - Clients cannot "break the abstraction" *no matter what they do*

Client vs. Implementer

- Provide a reusable interface without revealing implementation
  - You've been practicing this throughout 143 already
  - More difficult than it sounds due to aliasing and field-assignment (topic for today)

- We study concepts it in terms of ADTs instead of particular implementations in this class
  - Will use priority queues as our example in this lecture, but any ADT would do

Recall the abstraction

Clients:

"not trusted by ADT implementer"

- Can perform any sequence of ADT operations
- Can do anything type-checker allows on any accessible objects

```
new PQ(…)

insert(…)

deleteMin(…)

isEmpty()
```

Implementer:

- Should document how operations can be used and what is checked (raising appropriate exceptions)
  - E.g., parameter for method x not `null`
- If used correctly, correct priority queue for any client
- Client "cannot see" the implementation
  - E.g., binary min heap

Review: commenting

Let's practice our skills with the Client vs Implementer abstraction, through commenting

(look at code)

Commenting exercise: takeaways

- private comments for other coders looking at your file
- all public functionality should be commented for clients of your class
- implementation details should not be in public comments
- determine the line of abstraction, make sure you're not giving implementation details over that line

Coding Abstractions: our example

A priority queue with to-do items, so earlier dates "come first"

```
public class ToDoItem {
   … // some private fields (date, description)
   public void setDate(Date d) {…}
   public void setDescription(String d) {…}
   … // more methods
}
public class Date {
   … // some private fields (year, month, day)
   public int getYear() {…}
   public void setYear(int y) {…}
   … // more methods
}

// continued next slide…
```

Coding Abstractions: our example

A priority queue with to-do items, so earlier dates "come first"

```
public class ToDoPQ {
   … // some fields (array, size, …)
   public ToDoPQ() {…}
   void insert(ToDoItem t) {…}
   ToDoItem deleteMin() {…}
   boolean isEmpty() {…}
}
public class ToDoItem { … }
public class Date { … }
```

A mistake we taught you in 143

- Can you think of some more client code that might break the ToDoPQ?

```
public class ToDoPQ {
    // other fields
    public ToDoItem[] heap;
    // methods
    public ToDoPQ() {…}
    void insert(ToDoItem t) {…}
}

// client code:
pq = new ToDoPQ();
```

A mistake we taught you in 143

- Why we trained you to “mindlessly” make fields **`private`**:

```
public class ToDoPQ {
   … // other fields
   public ToDoItem[] heap; // problem!

    public ToDoPQ() {…}
   void insert(ToDoItem t) {…}
   …
}
// client:
pq = new ToDoPQ();
pq.heap = null;
pq.insert(…); // likely exception
```

- Today’s lecture: `private` does not solve all your problems!

Less obvious mistakes

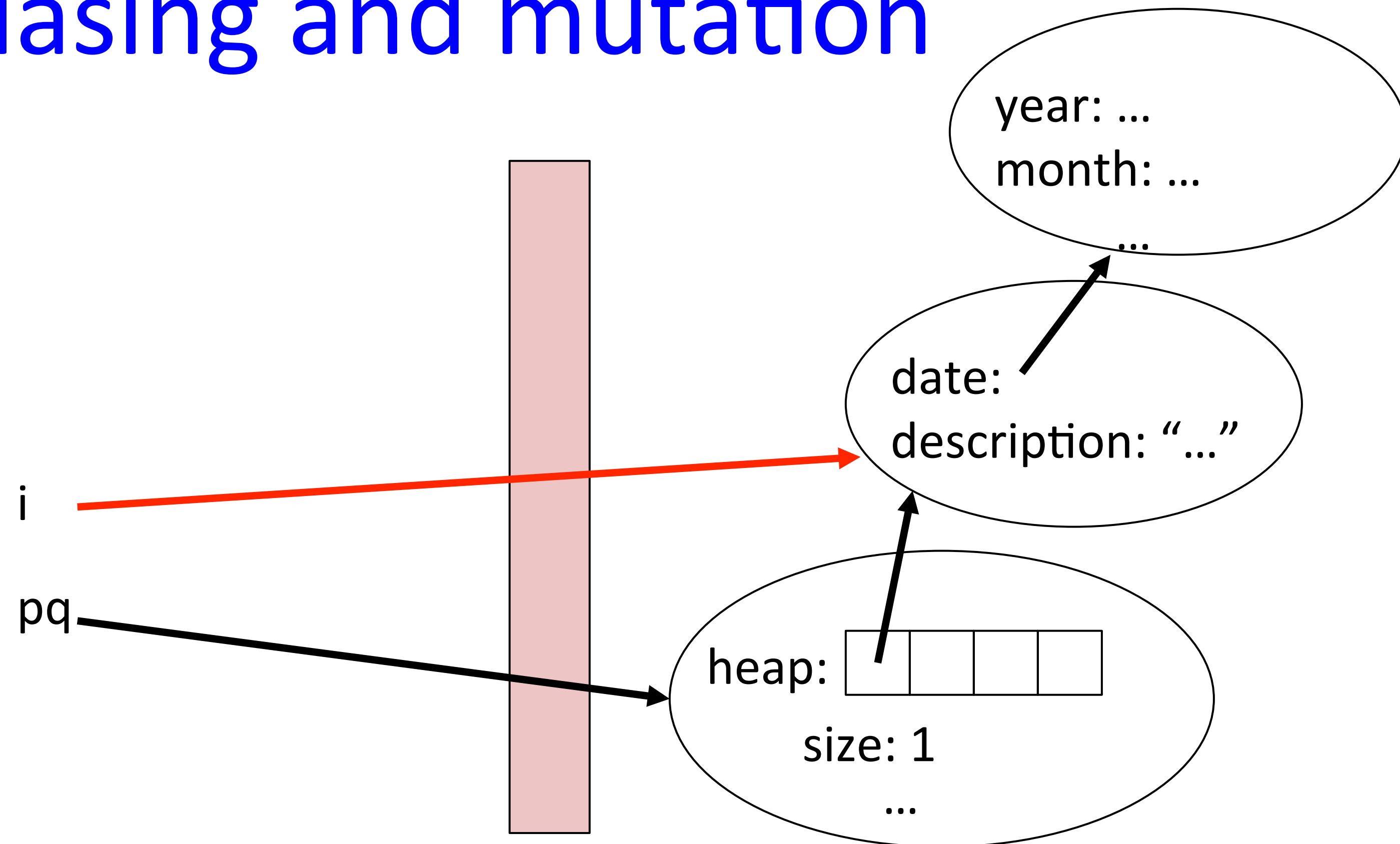
Can you think of some more client code that might break the ToDoPQ?

```
public class ToDoPQ {
   … // all private fields
   public ToDoPQ() {…}
   void insert(ToDoItem i) {…}
    ToDoItem deleteMin() {…}
   …
}

// client:
ToDoPQ   pq = new ToDoPQ();
ToDoItem i  = new ToDoItem(…);
pq.insert(i);
...
```

Less obvious mistakes

```
public class ToDoPQ {
   … // all private fields
   public ToDoPQ() {…}
   void insert(ToDoItem i) {…} // potential problem
   ToDoItem deleteMin() {…} // potential problem
   …
}

// client:
ToDoPQ   pq = new ToDoPQ();
ToDoItem i  = new ToDoItem(…);
pq.insert(i);
i.setDescription("some different thing");
pq.insert(i); // same object after update
x = deleteMin(); // x's description???
y = deleteMin(); // y's description???
```

Aliasing and mutation



- Client was able to update something inside the abstraction because client had an alias to it!
  - It is too hard to reason about and document what should happen, so better software designs avoid the issue!

More bad clients

What is wrong with this code? What is the date of the ToDoItem stored in variable x?

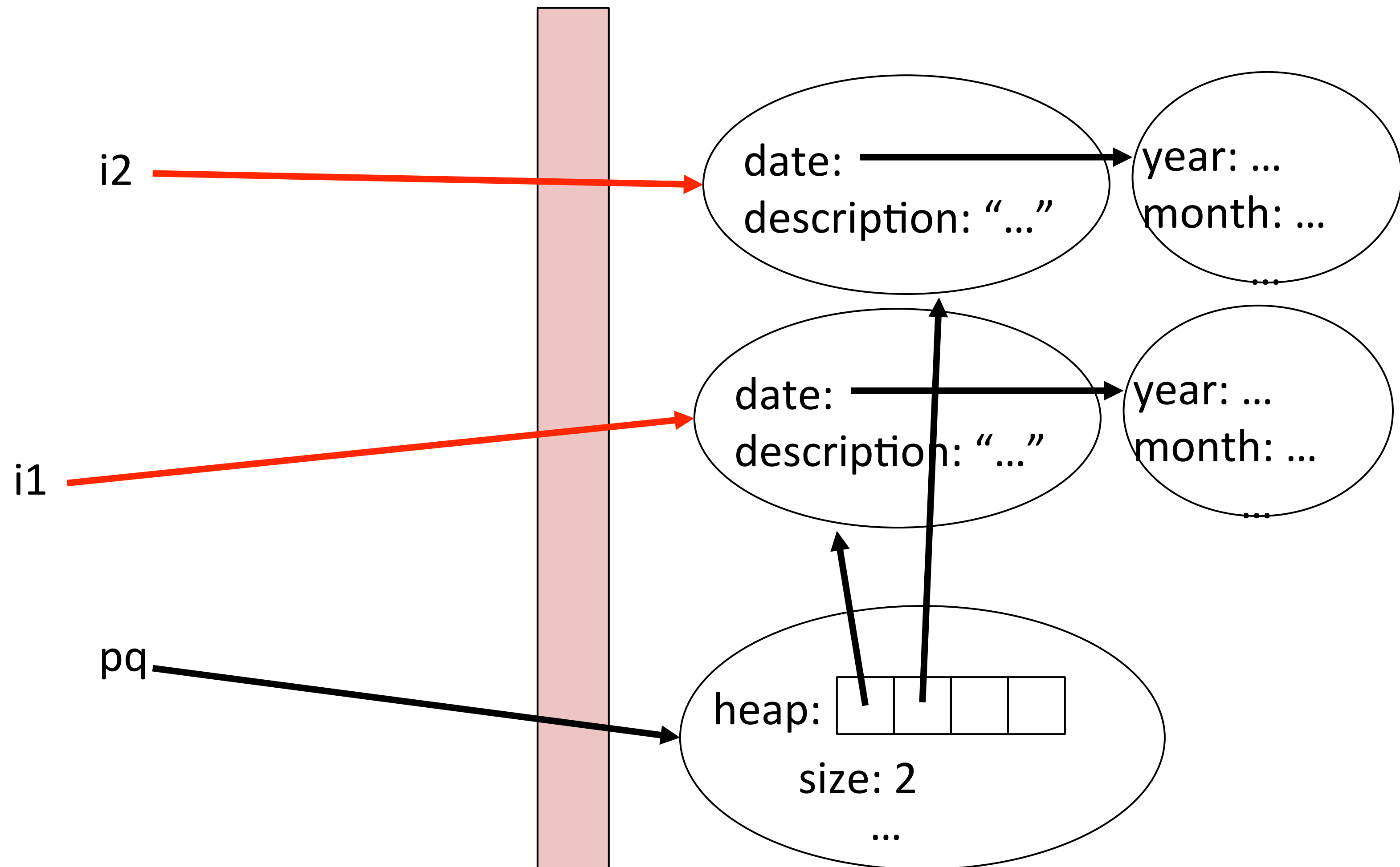
```
ToDoPQ   pq = new ToDoPQ();
ToDoItem i1 = new ToDoItem(…); // year 2013
ToDoItem i2 = new ToDoItem(…); // year 2014
pq.insert(i1);
pq.insert(i2);
i1.setDate(…); // year 2015
x = deleteMin();
```

More bad clients

What is wrong with this code? What is the date of the ToDoItem stored in variable x?

```
ToDoPQ   pq = new ToDoPQ();
ToDoItem i1 = new ToDoItem(…); // year 2013
ToDoItem i2 = new ToDoItem(…); // year 2014
pq.insert(i1);
pq.insert(i2);
i1.setDate(…); // year 2015
x = deleteMin(); // stores the data for i1, but
the date is now in year 2015
```

More bad clients



More bad clients

What is wrong with this client code? What happens when you compare the dates of i1 and i2 in order to do percolateUp when inserting?

```
pq = new ToDoPQ();
ToDoItem i1 = new ToDoItem(…);
pq.insert(i1);
i1.setDate(null);
ToDoItem i2 = new ToDoItem(…);
pq.insert(i2);
```

More bad clients

```
pq = new ToDoPQ();
ToDoItem i1 = new ToDoItem(…);
pq.insert(i1);
i1.setDate(null);
ToDoItem i2 = new ToDoItem(…);
pq.insert(i2); // NullPointerException
```

Get exception inside data-structure code even if `insert` did a careful check the first time that the date in the `ToDoItem` is not `null`

- Bad client later invalidates the check

The general fix

- Clients can't be trusted with pointers to your data.
- Avoid aliases into the internal data (the "red arrows") by copying objects as needed
  - Do not use the same objects inside and outside the abstraction because two sides do not know all mutation (field-setting) that might occur

A first attempt:

```
public class ToDoPQ {
    …
    void insert(ToDoItem i) {
      ToDoItem internal_i = i;
     }
}
```

Must copy the object

Notice this version accomplishes nothing

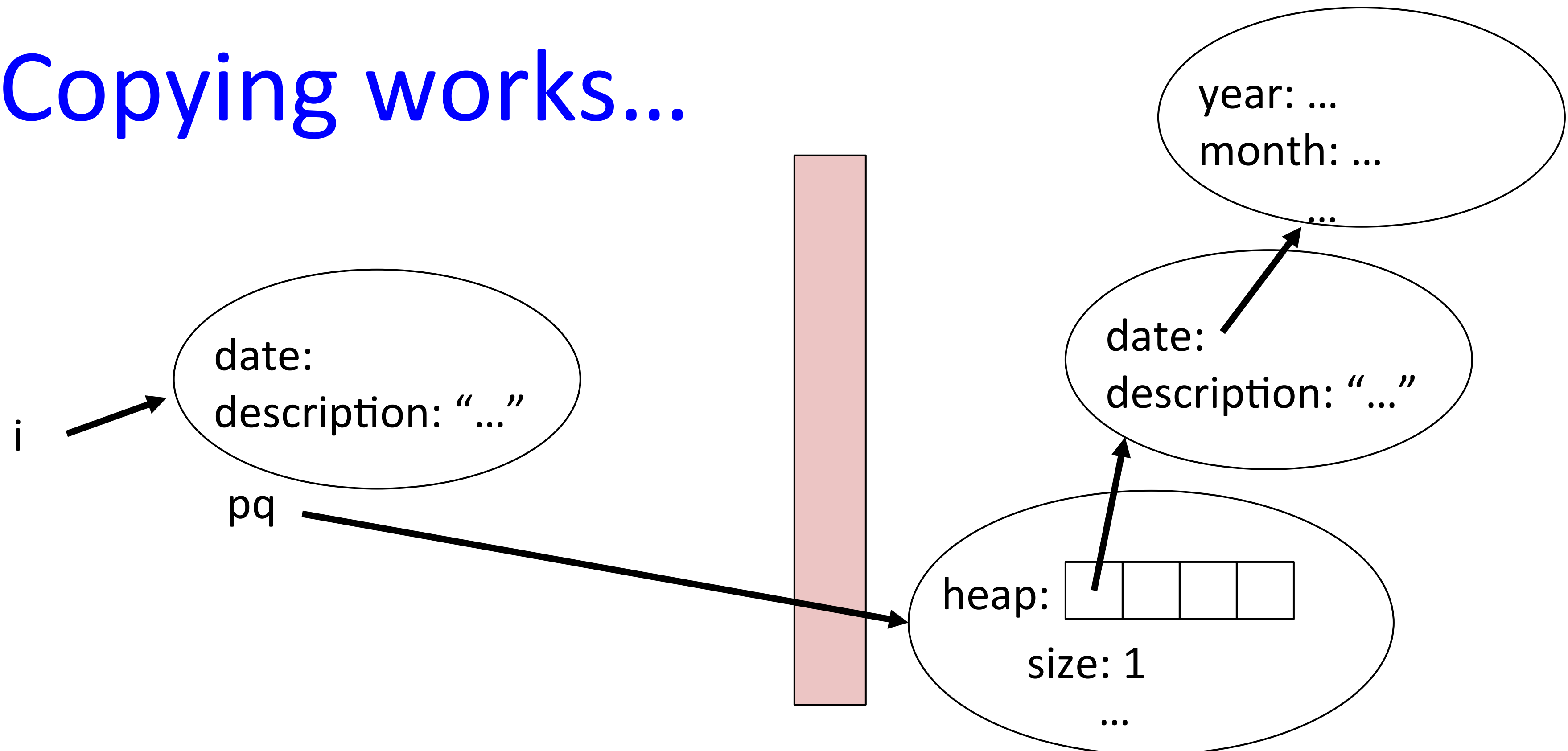
- Still the alias to the object we got from the client:

```
public class ToDoPQ {
    void insert(ToDoItem i) {
      ToDoItem internal_i = i;
      … // internal_i refers to same object
    }
}
```

second attempt:

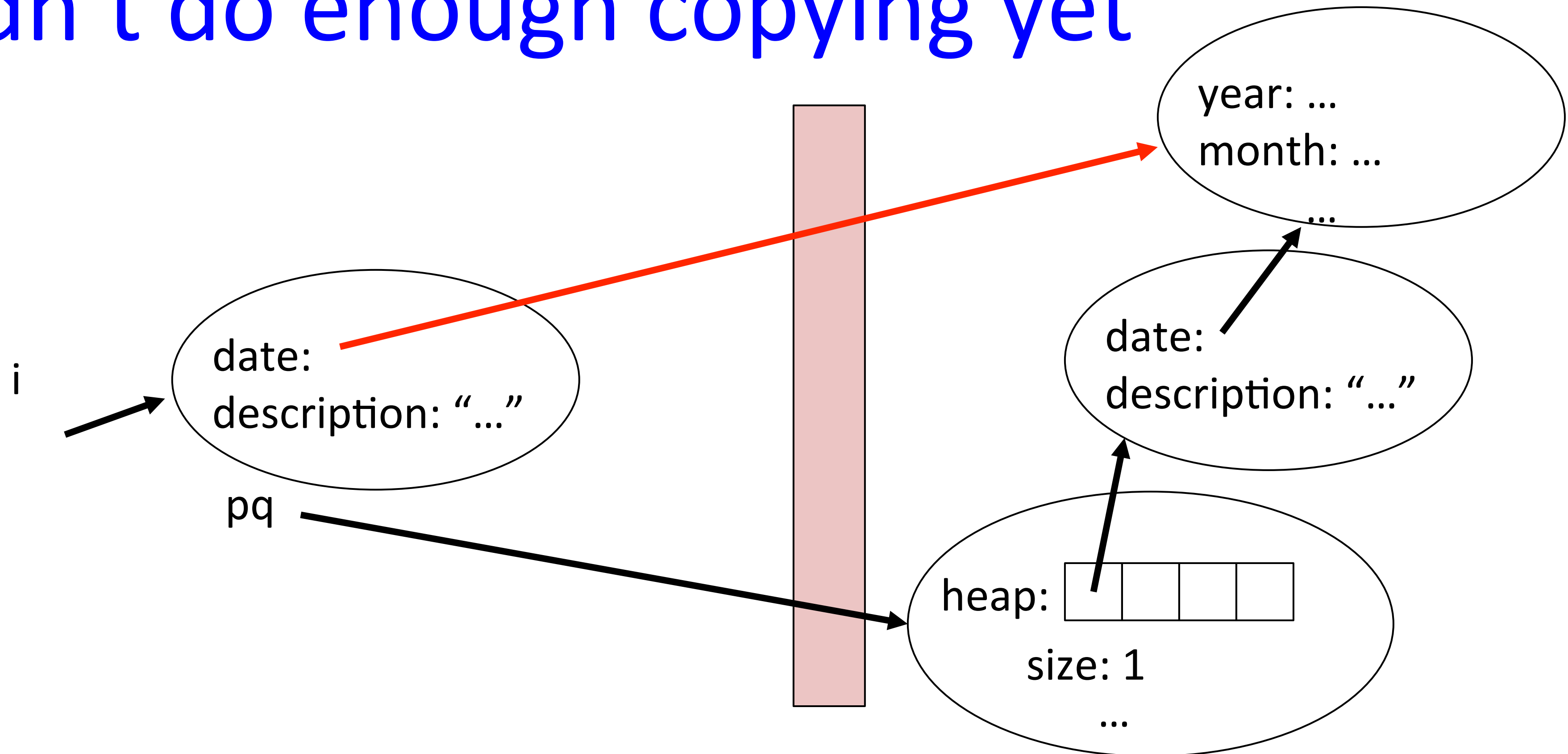
```
public class ToDoPQ {
    …
    void insert(ToDoItem i) {
      ToDoItem internal_i =
         new ToDoItem(i.date,i.description);
      … // use only the internal object
    }
}
```

Copying works…




```
ToDoItem i = new ToDoItem(…);
pq = new ToDoPQ();
pq.insert(i);
i.setDescription("some different thing");
pq.insert(i);
x = deleteMin();
y = deleteMin();
```

Didn't do enough copying yet




```
Date d = new Date(…)
ToDoItem i = new ToDoItem(d,"buy beer");
pq = new ToDoPQ();
pq.insert(i);
d.setYear(2015);
…
```

Deep copying (copy all the way down)

What if the client has an alias to i.date? Then depending on the implementation for ToDoItem, they may still have a reference to internal_i.date or internal_i.description.

```
public class ToDoPQ {
   void insert(ToDoItem i) {
     ToDoItem internal_i =
       new ToDoItem(i.date,i.description);
     … // use only the internal object
   }
}

public class ToDoItem {
    public ToDoItem(Date d, String desc) {
        this.d = new Date(d.year, d.month,
                          d.day);
        this.desc = desc;
    }
}
```

If you own all the objects being used, you can control the copying at every level. If you don't, then to deep copy, you have to copy everything.

Deep copying (copy all the things)

- For copying to work fully, usually need to also make copies of all objects referred to (and that they refer to and so on…)
  - All the way down to **`int`**, **`double`**, **`String`**, …
  - Called *deep copying* (versus our first attempt *shallow-copy*)
- Rule of thumb: Deep copy of things passed into abstraction

```
public class ToDoPQ {
   …
   void insert(ToDoItem i) {
     ToDoItem internal_i =
       new ToDoItem(new Date(…),
                    i.description);
     … // use only the internal object
   }
}
```

Constructors take input too

- General rule: Do not “trust” data passed to constructors
  - Check properties and make deep copies
- Example: Floyd’s algorithm for **`buildHeap`** should:
  - Check the array (e.g., for **`null`** values in fields of objects or array positions)
  - Make a deep copy: new array, new objects

```
public class ToDoPQ {
  // a second constructor that uses
  // Floyd's algorithm, but good design
  // deep-copies the array (and its contents)
  void PriorityQueue(ToDoItem[] items) {
     …
  }
}
```

That was copy-in, now copy-out…

- So we have seen:
  – Need to deep-copy data passed into abstractions to avoid pain and suffering

- Next:
  – Need to deep-copy data passed out of abstractions to avoid pain and suffering (unless data is "new" or no longer used in abstraction)

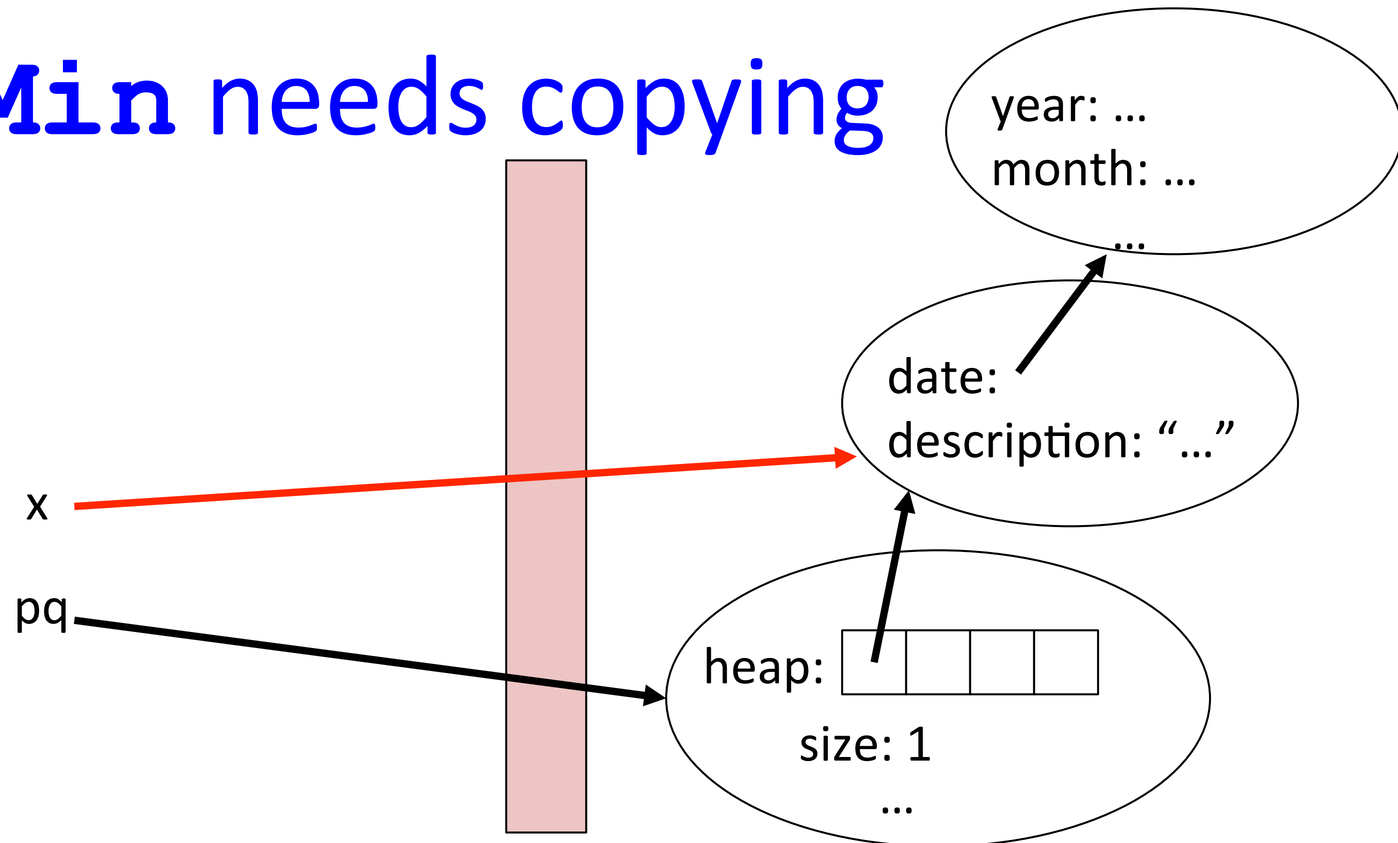
- Then:
  – If objects are immutable (no way to update fields or things they refer to), then copying unnecessary

`deleteMin` is fine

```
public class ToDoPQ {
   …
   ToDoItem deleteMin() {
     ToDoItem ans = heap[0];
     … // algorithm involving percolateDown
     return ans;
}
```

- Does not create an external alias because object returned is no longer part of the data structure
- Returns an alias to object that was in the heap, but now it is not, so conceptual "ownership" "transfers" to the client

getMin needs copying




```
ToDoItem i = new ToDoItem(…);
pq = new ToDoPQ();
x = pq.getMin();
x.setDate(…);
```

```
public class ToDoPQ {
  ToDoItem getMin() {
    int ans = heap[0];
    return ans;
  }
}
```

fixed: deep copy on return

- Just like we deep-copy objects from clients before adding to our data structure, we should deep-copy parts of our data structure and return the copies to clients

- Copy-in *and* copy-out

```
public class ToDoPQ {
  ToDoItem getMin() {
    ToDoItem ans = heap[0];
    return new ToDoItem(new Date(…),
                        ans.description);
  }
}
```

Less copying

- (Deep) copying is one solution to our aliasing problems

- Another solution is *immutability*
  - Make it so nobody can ever change an object or any other objects it can refer to (deeply)
  - Allows external aliases, but immutability makes them harmless

- In Java, a **`final`** field cannot be updated after an object is constructed, so helps ensure immutability
  - But **`final`** is a "shallow" idea and we need "deep" immutability

Immutability: This works

```
public class Date {
    private final int year;
    private final String month;
    private final String day;
}
public class ToDoItem {
    private final Date date;
    private final String description;
}
public class ToDoPQ {
    void insert(ToDoItem i){/*no copy-in needed!*/}
    ToDoItem getMin(){/*no copy-out needed!*/}
    …
}
```

Notes:

- `String` objects are immutable in Java
- (Using `String` for `month` and `day` is not great style though)

Immutability: This does not work

```
public class Date {
   private final int year;
   private String month; // not final
   private final String day;
   …
}
public class ToDoItem {
   private final Date date;
   private final String description;
}
public class ToDoPQ {
   void insert(ToDoItem i){/*no copy-in*/}
   ToDoItem getMin(){/*no copy-out*/}
   …
}
```

Client could mutate a `Date`'s `month` that is in our data structure

- So must do entire deep copy of `ToDoItem`

`final` is shallow

```
public class ToDoItem {
    private final Date date;
    private final String description;
}
```

- Here, **`final`** means no code can update the **`date`** or **`description`** fields after the object is constructed
- So they will always refer to the same **`Date`** and **`String`** objects
- But what if those objects have *their* contents change
  - Cannot happen with **`String`** objects
  - For **`Date`** objects, depends how we define **`Date`**
- So **`final`** is a "shallow" notion, but we can use it "all the way down" to get deep immutability

Immutability: This works

- When deep-copying, can “stop” when you get to immutable data
  - Copying immutable data is wasted work, so poor style

```
public class Date { // immutable
   private final int year;
   private final String month;
   private final String day;
   …
}
public class ToDoItem {
   private Date date;
   private String description;
}
public class ToDoPQ {
   ToDoItem getMin(){
    ToDoItem ans = heap[0];
    return new ToDoItem(ans.date, // okay!
                        ans.description);
   }
}
```

What about this?

```
public class Date { // immutable
    …
}
public class ToDoItem { // immutable (unlike last slide)
    …
}
public class ToDoPQ {
  // a second constructor that uses
  // Floyd's algorithm
  void PriorityQueue(ToDoItem[] items) {
     // what copying should we do?
     …
    }
}
```

What about this?

```
public class Date { // immutable
    …
}
public class ToDoItem { // immutable (unlike last slide)
    …
}
public class ToDoPQ {
  // a second constructor that uses
  // Floyd's algorithm
  void PriorityQueue(ToDoItem[] items) {
     // what copying should we do?
     …
    }
}
```

Copy the array, but do not copy the `ToDoItem` or `Date` objects

Today's Takeaways

- Client vs Implementer: what is the line of abstraction
- Copy-in and Copy-out to preserve abstraction and keep aliases from the client
- Deep copy and Immutability to keep your client from messing with your data

For future: Homework 4

- Won't be released until after the midterm
- You might choose to add to provided classes that make them not immutable
  - Leads to more copy-in-copy-out, but that's fine!
- *Or* you might leave them immutable and keep things in another dictionary (e.g., a **`HashMap`**)

There is more than one good design, but preserve your abstraction

- *Great practice with a key concept in software design*