Motivation

• Essential: knowing available data structures and their trade-offs
  – You’re taking a whole course on it! 😊

• However, you will rarely if ever re-implement these “in real life”
  – Provided by libraries

• But the key idea of an abstraction arises *all the time* “in real life”
  – Clients do not know how it is implemented
  – Clients do not need to know
  – Clients cannot “break the abstraction” *no matter what they do*
**Interface vs. implementation**

- Provide a reusable interface without revealing implementation

- More difficult than it sounds due to aliasing and field-assignment
  - Some common pitfalls

- So study it in terms of ADTs vs. data structures
  - Will use priority queues as example in lecture, but any ADT would do
  - Key aspect of grading your homework on graphs
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

Data structure:

```java
new PQ(...) 
insert(...) 
deleteMin(...) 
isEmpty()
```

- Should document how operations can be used and what is checked (raising appropriate exceptions)
  - E.g., fields not null
- If used correctly, correct priority queue for any client
- Client “cannot see” the implementation
  - E.g., binary min heap
Our example

- A priority queue with to-do items, so earlier dates “come first”
  - Simpler example than using Java generics
- Exact method names and behavior not essential to example

```java
public class Date {
    ... // some private fields (year, month, day)
    public int getYear() {...}
    public void setYear(int y) {...}
    ... // more methods
}
public class ToDoItem {
    ... // some private fields (date, description)
    public void setDate(Date d) {...}
    public void setDescription(String d) {...}
    ... // more methods
}
// continued next slide...
```
Our example

- A priority queue with to-do items, so earlier dates “come first”
  - Simpler example than using Java generics
- Exact method names and behavior not essential to example

```java
public class Date { ... }
public class ToDoItem { ... }
public class ToDoPQ {
    ... // some private fields (array, size, ...)
    public ToDoPQ() {...}
    void insert(ToDoItem t) {...}
    ToDoItem deleteMin() {...}
    boolean isEmpty() {...}
}
```
An obvious mistake

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

  ```java
  public class ToDoPQ {
      ... // other fields
      public ToDoItem[] heap;
      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!
  - Upcoming pitfalls can occur even with all `private` fields
Less obvious mistakes

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

    // 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

```java
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(); // “wrong” (???) item?
    // What date does returned item have???
```
More bad clients

pq

heap: size: 2

i1

i2

date: description: “…”
year: …
month: …

date: description: “…”
year: …
month: …

...
More bad clients

```java
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 that the date in the `ToDoItem` is not `null`
- Bad client later invalidates the check
The general fix

- 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
  - “Copy-in-copy-out”

- A first attempt:

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

public class ToDoPQ {
    ...
    void insert(ToDoItem i) {
        ToDoItem internal_i =
            new ToDoItem(i.date, i.description);
        ...
        // use only the internal 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
    }
}
```java
ToDoItem i = new ToDoItem(...);
pq = new ToDoPQ();
pq.insert(i);
i.setDescription("some different thing");
pq.insert(i);
```
Date d = new Date(...)  
ToDoItem i = new ToDoItem(d, "buy beer");  
pq = new ToDoPQ();  
pq.insert(i);  
d.setYear(2015);  
...
Deep copying

- 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

```java
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

```java
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

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

- Does not create a “red arrow” 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
public class ToDoPQ {
    ToDoItem getMin() {
        int ans = heap[0];
        return ans;
    }
}

- Uh-oh, creates a “red arrow”

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

getMin needs copying
The fix

• 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

```java
public class ToDoPQ {
    ToDoItem getMin() {
        int 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 “red arrows”, 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
This works

```java
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)
This does not work

```java
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**

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

- Here, `final` means no code can update the `year` 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.
This works

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

```java
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() {
        int ans = heap[0];
        return new ToDoItem(ans.date, // okay!
                            ans.description);
    }
}
```
What about this?

```java
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
Homework 5

- You are implementing a graph abstraction
- As provided, `Vertex` and `Edge` are immutable
  - But `Collection<Vertex>` and `Collection<Edge>` are not
- You might choose to add fields to `Vertex` or `Edge` 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 like “best-path-cost-so-far” 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