Initial design

There's not a lot of room for creativity. This design is pretty much ideal:

```java
class TodoItem {
  enum TodoStatus { DONE, NOT_DONE }

  private String desc;
  private TodoStatus status;

  // getters, setters
}

class TodoList {
  private List<TodoItem> items;

  void add(TodoItem);
  void remove(TodoItem);
}
```

Some points of variation:

- Adding IDs to all entities (though see the "Substantially different architectures" under the mistakes below)
- Adding an extra timestamp or user parameter to every function
- Using immutable data structures

Some common mistakes:

- Using a boolean instead of an enum for `TodoStatus`. Booleans are less open than enums; adding new statuses (a totally predictable change) will require rewriting.
  - They are also less descriptive. Because `setDoneStatus(true)` is unclear, some people hence create a restricted `markDone()` method to compensate.
- Putting the methods to modify `TodoItem's` on `TodoList` rather than `TodoItem`
- Having a getter that returns the underlying mutable list of `TodoList`. A method to get the list of `TodoItem's` can exist, but it cannot allow clients to assume that modifying said list would modify the original `TodoList`

Substantially different architectures:

- The only one I've seen are variants of doing everything in terms of IDs, e.g.: `TodoList` has an `add(int)` method instead of `add(TodoItem)`.
- There's another variant still where there is a single `TodoApi` structure with functions like `addItemToList(int listId, int itemId)`.
- These definitions don't make sense without an underlying relational model. This API is substantially harder to program against compared to the recommended datatype-oriented one

- If you use different types for each kind of ID, it becomes equivalent to the datatype-oriented version, but with extra indirection. See http://www.pathsensitive.com/2018/07/when-your-data-model-means-something.html (section: "The True Relation") for a discussion of the correspondence between them.
- Not having a separate type for `TodoList`, and just using `List<TodoItem>`. Code that uses it will need to be rewritten as soon as you want to add any extra data/functionality to the todo list itself.
- Note that, in a language like C, Haskell, or OCaml with type aliases / typedefs, just writing `type TodoList = [TodoItem]` and using it in lieu of the raw type can go a long way towards making it easier to add additional information to `TodoList's` later.

Round 1

Adding stickers/more statuses/due dates/other attributes. Supporting sorting/filtering.

```
class TodoItem {
  enum TodoStatus { DONE, NOT_DONE, ... }
  // other fields same as above

  enum Priority { URGENT, NORMAL, LO_PRI, ... }

  private @Nullable Sticker sticker.
  private @Nullable Date dueDate;
  private Priority priority;
  // other attributes

  // getters, setters
}

class TodoList {
  // other fields same as before

  public List<TodoItem> sort(Comparator<TodoItem> c);
  public List<TodoItem> filter(Function<TodoItem, Boolean) f);
}
```

Some common mistakes:

- Using a fixed language of filters/comparisons instead of a lambda
  - E.g.: `filterByStatus(TodoStatus)`
  - E.g.: Creating custom filter
  - In Week 5, we discuss defunctionalization/refunctionalization. This fixed language is the defunctionalized form of using a lambda
    - ⋆ It thus has the same tradeoffs: the defunctionalized form is more inspectable, the lambda form (refunctionalized) is more open
  - There isn't really much reason to serialize a filter function, and this approach takes much more work and is more restricted than the lambda
- Other representations of priorities.

    - The ideal is to just put in your code ”the priorities form an ordered set,” which the solution above accomplishes.
    - Using raw integers (or, worse, raw integers of bounded range) makes it harder to add new priorities between other priorities. but using a fixed set of floating-point (or, better yet, rational) numbers does allow for arbitrary extensibility.
- Stringly-typed code, e.g.: `private String sticker`
    - This bakes in the idea that a sticker contains no information other than the name.
    - It makes it harder to add extra features to stickers later than need additional information.
    - This is similar to the `AttackInfo` example in the Week 6 lecture.

Round 2

Adding users, friends, and public/private todo items.

This is where many people start to go wrong. If you bake privacy into the representation, perhaps using something like the below, then you'll dodge the potential privacy bugs in the next section without even realizing they exist. On the other hand, if you just slap an `isPrivate` field onto `TodoItem's` and let the user interface filter them out, then you'll either have lots of privacy bugs or a big mess.

```java
class User {
  private GUID id;
  private Set<User> friends;
  private TodoList todoList;
}

class TodoItem {
  // ... other fields

  enum PrivacyStatus { PRIVATE, FRIENDS }

  private PrivacyStatus privacy;
}

class TodoList {
  // Remove all getters/setters
  public class TodoListView {
    // Add the getters here
  }
  public class TodoListEditor {
    // Add the setters here
  }

  public TodoListView viewFor(User viewer);
  public TodoListEditor edit(User editor);
}
```

Some variations:

- Many other ways of enforcing the separation between own and viewed todo lists. E.g.: separate `OwnTodoList` and `ReadOnlyTodoList` types.
- Using raw ints for IDs. As long as they're generated uniquely and most code does not touch IDs, there's little danger in doing so.
- Conversely: This design doesn't need IDs; it just needs fast equality-checks on users.
- Many also add IDs to the other entities here:
  - One option: Having distinct ID types for each entity (ItemId, ListId, etc)
- Using a `PrivacyStatus` interface instead of a boolean/enum. This is the refunctionalized form of the enum.
  - This will be necessary if you want to add complex privacy controls but is of little benefit before.
  - Whether a `PrivacyStatus` is an enum or anything that implements an interface, the API can be the same, so it's easy to do this in the future.

Some common mistakes:

- Using a boolean for privacy status. This is a pretty minor problem but has the same drawbacks as using a boolean for done status in the initial design.

Round 3:

Your choice of:

1) Counting all todo items on the screen
2) Viewing what a todo list looked like at a given point in time

The purpose of this round (really, of this entire design exercise) is to lure you into making a privacy violation.

- For (1), you must exclude private items when making a count, else it will leak information

  - This can be particularly bad if the user interface is displaying a filtered view of a friend's list, but
  - the UI displays a count based on the underlying list.

- For (2), going back in time may allow a viewer to see items that are currently marked private but weren't initially

The implementation of (1) is straightforward code atop the existing data structures and APIs.

- If you followed a variation of the recommended designs from part 2, this is trivial.
- If you require all accessors of lists to add privacy checks, then your implementations must do these privacy checks.
- For a real example of failing to do a very similar privacy check, see this bug in Facebook: https://twitter.com/jimmykoppel/status/1459278887393718274

Implementing (2) is much more complicated. Three families of approaches:

1) Store copies of each item for each timestamp changed; the viewer then searches by timestamp. This makes privacy and efficiency difficult.

2) Use a global change log. The system state is snapshotted. Each update in the system pushes an Action into the log. This can then be partially replayed. This is more efficient, but the only way I see to get privacy is to replay the state backwards from the present, and have special casing for undoing a markPrivate action.

3) Change all entities to be immutable; add a timestamp and parent to each. E.g.:

```java
public class TodoItem {
  private DateTime timestamp;
  private @Nullable TodoItem parent;
  private TodoStatus status;
  // ...


  public TodoItem withUpdatedStatus(TodoStatus); //
   ↪  returnValue.parent == this; returnValue has timestamp set to
   ↪  now
}
```

Adding/removing an element to a `TodoList` will do a similar immutable update to what `withUpdatedStatus` does.

However, changing a `TodoItem` will mutate its containing lists in place. Then, the latest version will always be used for privacy checks. (Can implement this by placing a mutable `TodoItemRef` in the lists, rather than a `TodoItem` directly.) This solution is essentially a variant of #2, using a tree-structured log instead of a list. It can be represented as a changelog rather than a list of successive versions as an optimization.

This approach is the cleanest solution I've found to this problem, and the only one which handles privacy concerns with no explicit code.