L9: Abstract Data Types Part 1

Today
- Abstract data types
- Representation independence
- Representation exposure
- Invariants

Required Reading
- Review previous readings on: access control, classes and objects, static and final keywords

Introduction
In this lecture, we look at a powerful idea, abstract data types, which enable us to separate how we use a data structure in a program from the particular form of the data structure itself. Abstract data types address a particularly dangerous dependence, that of a client of a type on the type’s representation. We’ll see why this is dangerous and how it can be avoided. We’ll also discuss the classification of operations, and some principles of good design for abstract data types.

What Abstraction Means
Abstract data types are an instance of a general principle in software engineering, which goes by many names with slightly different shades of meaning. Here are some of the names that are used for this idea:

- **Abstraction.** Omitting or hiding low-level details with a simpler, higher-level idea.
- **Modularity.** Dividing up a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.
- **Encapsulation.** Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can’t damage its integrity.
- **Information hiding.** Hiding details of a module’s implementation from the rest of the system, so that those details can be changed later without changing the rest of the system.
- **Separation of concerns.** Making a feature (or “concern”) the responsibility of a single module, rather than spreading it across multiple modules.

As a software engineer, you should know these terms, because you will run into them frequently. The fundamental purpose of all of these ideas is to help achieve the three important properties that we care about in 6.005: safety from bugs, ease of understanding, and readiness for change.

User-Defined Types
In the early days of computing, a programming language came with built-in types (such as integers, booleans, strings, etc.) and built-in procedures, e.g. for input and output. Users could define their own procedures: that’s how large programs were built.
A major advance in software development was the idea of abstract types: that one could design a programming language to allow user-defined types, too. This idea came out of the work of many researchers, notably Dahl (the inventor of the Simula language), Hoare (who developed many of the techniques we now use to reason about abstract types), Parnas (who coined the term information hiding and first articulated the idea of organizing program modules around the secrets they encapsulated), and here at MIT, Barbara Liskov and John Guttag, who did seminal work in the specification of abstract types, and in programming language support for them – and developed 6.170, the predecessor to 6.005. In 2010, Barbara Liskov earned the Turing Award, computer science’s equivalent of the Nobel Prize, for her work on abstract types.

The key idea of data abstraction is that a type is characterized by the operations you can perform on it. A number is something you can add and multiply; a string is something you can concatenate and take substrings of; a boolean is something you can negate, and so on. In a sense, users could already define their own types in early programming languages: you could create a record type date, for example, with integer fields for day, month and year. But what made abstract types new and different was the focus on operations: the user of the type would not need to worry about how its values were actually stored, in the same way that a programmer can ignore how the compiler actually stores integers. All that matters is the operations.

In Java, as in many modern programming languages, the separation between built-in types and user-defined types is a bit blurry. The classes in java.lang, such as Integer and Boolean are built-in; whether you regard all the collections of java.util as built-in is less clear (and not very important anyway). Java complicates the issue by having primitive types that are not objects. The set of these types, such as int and boolean, cannot be extended by the user.

### Classifying Types and Operations

Types, whether built-in or user-defined, can be classified as mutable or immutable. The objects of a mutable type can be changed: that is, they provide operations which when executed cause the results of other operations on the same object to give different results. So Date is mutable, because you can call setMonth and observe the change with the getMonth operation. But String is immutable, because its operations create new string objects rather than changing existing ones. Sometimes a type will be provided in two forms, a mutable and an immutable form. StringBuilder, for example, is a mutable version of String (although the two are certainly not the same Java type, and are not interchangeable).

The operations of an abstract type are classified as follows:

- **Creators** create new objects of the type. A creator may take an object as an argument, but not an object of the type being constructed.
- **Producers** create new objects from old objects of the type. The concat method of String, for example, is a producer: it takes two strings and produces a new one representing their concatenation.
- **Mutators** change objects. The add method of List, for example, mutates a list by adding an element to the end.
- **Observers** take objects of the abstract type and return objects of a different type. The size method of List, for example, returns an integer.

We can summarize these distinctions schematically like this:

- **creator**: $t^* \rightarrow T$
- **producer**: $T+, t^* \rightarrow T$
- **mutator**: $T+, t^* \rightarrow \text{void}$
observer: \( T^+, t^* \rightarrow t \)

These show informally the shape of the signatures of operations in the various classes. Each \( T \) is the abstract type itself; each \( t \) is some other type. In general, when a type is shown on the left, it can occur more than once. For example, a producer may take two values of the abstract type; string \texttt{concat} takes two strings. The occurrences of \( t \) on the left may also be omitted; some observers take no non-abstract arguments (e.g., \texttt{size}), and some take several.

Here are some examples of abstract data types, along with their operations:

\texttt{int} is Java’s primitive integer type. \texttt{int} is immutable, so it has no mutators.
- creators: the numeric literals 0, 1, 2, …
- producers: arithmetic operators \(+, - , \times, \div\)
- observers: comparison operators \(==, !=, <, >\)
- mutators: none (it’s immutable)

\texttt{List} is Java’s list interface. \texttt{List} is mutable. \texttt{List} is also an \textit{interface}, which means that other classes provide the actual implementation of the data type. These classes include \texttt{ArrayList} and \texttt{LinkedList}.
- creators: \texttt{ArrayList} and \texttt{LinkedList} constructors, \texttt{Collections.singletonList()}
- producers: \texttt{Collections.unmodifiableList()}
- observers: \texttt{size()}, \texttt{get()}
- mutators: \texttt{add()}, \texttt{remove()}, \texttt{addAll()}, \texttt{Collections.sort()}

\texttt{String} is Java’s string type. \texttt{String} is immutable.
- creators: \texttt{String()}, \texttt{String(char[])} constructors
- producers: \texttt{concat()}, \texttt{substring()}, \texttt{toUpperCase()}
- observers: \texttt{length()}, \texttt{charAt()}
- mutators: none (it’s immutable)

This classification gives some useful terminology, but it’s not perfect. In complicated data types, there may be an operation that is both a producer and a mutator, for example. Some people use the term \textit{producer} to imply that no mutation occurs.

**Designing an Abstract Type**

Designing an abstract type involves choosing good operations and determining how they should behave. A few rules of thumb:

It’s better to have \textbf{a few, simple operations} that can be combined in powerful ways, rather than lots of complex operations.

Each operation should have a well-defined purpose, and should have a \textbf{coherent} behavior rather than a panoply of special cases. We probably shouldn’t add a \textit{sum} operation to \texttt{List}, for example. It might help clients who work with lists of \texttt{Integers}, but what about lists of \texttt{Strings}? Or nested lists? All these special cases would make \textit{sum} a hard operation to understand and use.
The set of operations should be **adequate**; there must be enough to do the kinds of computations clients are likely to want to do. A good test is to check that every property of an object of the type can be extracted. For example, if there were no `get` operation, we would not be able to find out what the elements of a list are. Basic information should not be inordinately difficult to obtain. The `size` method is not strictly necessary for `List`, because we could apply `get` on increasing indices until we get a failure, but this is inefficient and inconvenient.

The type may be generic: a list or a set, or a graph, for example. Or it may be domain-specific: a street map, an employee database, a phone book, etc. But it **should not mix generic and domain-specific features**. A `Deck` type intended to represent a sequence of playing cards shouldn’t have a generic `add` method that accepts arbitrary objects (like integers or strings). Conversely, it wouldn’t make sense to put a domain-specific method like `dealCards` into the generic type `List`.

### Representation Independence

A good abstract data type should be **representation independent**. This means that the use of an abstract type is independent of its representation (the actual data structure or data fields used to implement it), so that changes in representation have no effect on code outside the abstract type itself. For example, the operations offered by `List` are independent of whether the list is represented as a linked list or as an array.

You won’t be able to change the representation of an ADT at all unless its operations are fully specified with preconditions (requires), postconditions (effects), and frame conditions (modifies), so that clients know what to depend on, and you know what you can safely change.

### Example: Different Representations for Strings

Let’s look at a simple abstract datatype to see what representation independence means and why it’s useful. The `MyString` type below has far fewer operations than the real Java `String`, and their specs are a little different, but it’s still illustrative. Here are the specs for the ADT:

```java
public class MyString {

    /////////////////////// Example of a creator operation///////////////////
    /** @param b a boolean value  
     * @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) {
        ... 
    }

    /////////////////////// Examples of observer operations ///////////////
    /** @return number of characters in this string */
    public int length() {
        ... 
    }

    /** @param i character position (requires 0 <= i < string length)  
     * @return character at position i */
    public char charAt(int i) {
        ... 
    }

    /////////////////////// Example of a producer operation/////////////////
    /** Get the substring between start (inclusive) and end (exclusive).  
     * @param start starting index  
     * @param end ending index. Requires 0 <= start <= end <= string length.  
     * @return string consisting of charAt(start)...charAt(end-1) */
    public MyString substring (int start, int end) {
        ... 
    }
}
```

```
These public operations and their specifications are the only information that a client of this datatype is allowed to know about. Following the test-first programming paradigm, in fact, the first client we should create is a test suite that exercises these operations according to their specs. At the moment, we don’t even have an equality operation defined on these MyStrings. We’ll talk about how to implement equality carefully in a future lecture, but for now, the only operations we can perform with MyStrings are the ones we’ve defined above: valueOf, length, charAt, substring. So our tests have to limit themselves to those operations. For example, here’s one test for the valueOf operation:

```java
MyString s = MyString.valueOf(true);
assertEquals(4, s.length());
assertEquals('t', s.charAt(0));
assertEquals('r', s.charAt(1));
assertEquals('u', s.charAt(2));
assertEquals('e', s.charAt(3));
```

We’ll come back to the question of testing ADTs in the next section. For now, let’s look at a simple representation for MyString: just an array of characters, exactly the length of the string (no extra room at the end). Here’s how that internal representation would be declared, as an instance variable within the class:

```java
private char[] a;
```

With that choice of representation, the operations would be implemented in a straightforward way:

```java
public static MyString valueOf(boolean b) {
  MyString s = new MyString();
  s.a = b ? new char[] { 't', 'r', 'u', 'e' } :
           new char[] { 'f', 'a', 'l', 's', 'e' };
  return s;
}

public int length() {
  return a.length;
}

public char charAt(int i) {
  return a[i];
}

public MyString substring(int start, int end) {
  MyString that = new MyString();
  that.a = new char[end - start];
  System.arraycopy(this.a, start, that.a, 0, end - start);
  return that;
}
```

(Why don’t charAt() and substring() have to check whether their parameters are within the valid range? What do you think will happen if the client calls these implementations with illegal inputs?)

But we’re giving up an opportunity for performance improvement here. Because this datatype is immutable, the substring() operation doesn’t really have to copy characters out into a fresh array. It could just point to the original string’s character array and keep track of the start and end that the new substring object represents. (The actual String implementation in Java does this.) So we could change the internal representation of this class to:

```java
private char[] a;
private int start;
private int end;
```

With this new representation, the operations are now implemented like this:
public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] {'t', 'r', 'u', 'e'}
      : new char[] {'f', 'a', 'l', 's', 'e'};
    s.start = 0;
    s.end = s.a.length;
    return s;
}

public int length() {
    return end - start;
}

public char charAt(int i) {
    return a[start + i];
}

public MyString substring (int start, int end) {
    that.a = this.a;
    that.start = this.start + start;
    that.end = this.start + end;
    return that;
}

Testing an Abstract Datatype

We build a test suite for an abstract datatype by creating tests for each of its operations. These tests inevitably interact with each other, since the only way to test creators, producers, and mutators is by calling observers on the objects that result.

Here’s how we might partition the input spaces of the four operations in our MyString type:

// testing strategy for each operation of MyString:
//
// valueOf():
//  true, false
// length():
//  string len = 0, 1, n
//  string = produced by valueOf(), produced by substring()
// charAt():
//  string len = 1, n
//  i = 0, middle, len-1
//  string = produced by valueOf(), produced by substring()
// substring():
//  string len = 0, 1, n
//  start = 0, middle, len
//  end = 0, middle, len
//  end-start = 0, n
//  string = produced by valueOf(), produced by substring()

Then a compact test suite that covers all these partitions might look like:

@Test public void testValueOfTrue() {
    MyString s = MyString.valueOf(true);
    assertEquals(4, s.length());
    assertEquals('t', s.charAt(0));
    assertEquals('r', s.charAt(1));
    assertEquals('u', s.charAt(2));
    assertEquals('t', s.charAt(3));
}
```java
@Test public void testValueOfFalse() {
    MyString s = MyString.valueOf(false);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}
```

```java
@Test public void testEndSubstring() {
    MyString s = MyString.valueOf(true).substring(2, 4);
    assertEquals(2, s.length());
    assertEquals('u', s.charAt(0));
    assertEquals('e', s.charAt(1));
}
```

```java
@Test public void testMiddleSubstring() {
    MyString s = MyString.valueOf(false).substring(1, 2);
    assertEquals(1, s.length());
    assertEquals('a', s.charAt(0));
}
```

```java
@Test public void testSubstringIsWholeString() {
    MyString s = MyString.valueOf(false).substring(0, 5);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}
```

```java
@Test public void testSubstringOfEmptySubstring() {
    MyString s = MyString.valueOf(false).substring(1, 1).substring(0, 0);
    assertEquals(0, s.length());
}
```

### Rep Independence and Testing

The idea of representation independence may seem murky. What does it mean, concretely? One formal notion is contextual equivalence. Basically, two implementations of an abstract data type are contextually equivalent if we can think of them as black boxes, hit them with arbitrary sequences of method calls, and get the same results. The hypothetical program running all these method calls is the context. If we implement the same ADT in two different ways, we can make that notion real by running some tests designed to detect differences in behavior between our two implementations. The more such tests succeed, the more confidence we have that both classes hide their representations.

### Preserving Invariants

Resuming our discussion of what makes a good abstract datatype, the final, and perhaps most important, property of a good abstract data type is that it preserves its own invariants. An invariant is a property of a program that is always true. Immutability is one crucial invariant that we’ve already
encountered: once created, an immutable object should always represent the same value, for its entire lifetime.

When an ADT preserves its own invariants, reasoning about the code becomes much easier. If you can count on the fact that Strings never change, you can rule out that possibility when you’re debugging code that uses Strings — or when you’re trying to establish an invariant for another ADT. Contrast that with a string class that guarantees that it will be immutable only if its clients promise not to change it. Then you’d have to check all the places in the code where the string might be used.

Immutability

We’ll see many interesting invariants. Let’s focus on immutability for now. Here’s a specific example:

```java
/**
 * This immutable datatype represents a tweet from Twitter.
 */
public class Tweet {
  public String author;
  public String text;
  public Date timestamp;

  /**
   * Make a Tweet.
   * @param author  Twitter user who wrote the tweet.
   * @param text    text of the tweet
   * @param timestamp date/time when the tweet was sent
   */
  public Tweet(String author, String text, Date timestamp) {
    this.author = author;
    this.text = text;
    this.timestamp = timestamp;
  }
}
```

How do we guarantee that Tweet objects are immutable — that, once a tweet is created, its author, message, and date can never be changed?

The first threat to immutability comes from the fact that clients can (in fact, must!) directly access its fields. So nothing’s stopping us from writing code like this:

```java
Tweet t = new Tweet("justinbieber",
  "Thanks to all those beliebers out there inspiring me every day",1
  new Date());
  t.author = "rbmllr";
```

This is a trivial example of representation exposure, meaning that code outside the class can modify the representation directly. Rep exposure like this threatens not only invariants, but also representation independence. We can’t change the implementation of Tweet without affecting all the clients who are directly accessing those fields.

Fortunately, Java gives us language mechanisms to deal with this kind of rep exposure:

```java
public class Tweet {
  private final String author;
```

---

private final String text;
private final Date timestamp;

public Tweet(String author, String text, Date timestamp) {
    this.author = author;
    this.text = text;
    this.timestamp = timestamp;
}

/** @return Twitter user who wrote the tweet */
public String getAuthor() {
    return author;
}

/** @return text of the tweet */
public String getText() {
    return text;
}

/** @return date/time when the tweet was sent */
public Date getTimestamp() {
    return timestamp;
}
}

The private and public keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class. The final keyword also helps by guaranteeing that the fields of this immutable type won’t be reassigned after the object is constructed.

But that’s not the end of the story: the rep is still exposed! Consider this (perfectly reasonable) client code that uses Tweet:

    /** @return a tweet that retweets t, one hour later*/
    public static Tweet retweetLater(Tweet t) {
        Date d = t.getTimestamp();
        d.setHours(d.getHours()+1);
        return new Tweet("rbmllr", t.getText(), d);
    }

retweetLater takes a tweet and should return another tweet with the same message (called a retweet) but sent an hour later. The retweetLater method might be part of a system that automatically echoes funny things that Twitter celebrities say.

What’s the problem here? The getTimestamp call returns a reference to the same date object referenced by tweet t. t.timestamp and d are aliases to the same mutable object. So when that date object is mutated by d.setHours(), this affects the date in t as well:

Tweet’s immutability invariant has been broken. The problem is that Tweet leaked out a reference to a mutable object that its invariant depended on. We exposed the rep, in such a way that Tweet can no longer guarantee that its objects are immutable. Perfectly reasonable client code created a subtle bug.
We can patch this kind of rep exposure by defensive copying: making a copy of a mutable object to avoid leaking out references to the rep. Here’s the code:

```java
public Date getTimestamp() {
    return new Date(Date.getTime());
}
```

Mutable types often have a copy constructor that allows you to make a new instance that duplicates the value of an existing instance. In this case, Date’s copy constructor uses the timestamp value, measured in seconds since January 1, 1970. As another example, StringBuilder’s copy constructor takes a String. Another way to copy a mutable object is clone(), which is supported by some types but not all. There are unfortunate problems with the way clone() works in Java. For more, see Josh Bloch, Effective Java, item 10.

So we’ve done some defensive copying in the return value of getTimestamp. But we’re not done yet! There’s still rep exposure. Consider this (again perfectly reasonable) client code:

```java
/** @return a list of 24 inspiring tweets, one per hour today */
public static List<Tweet> tweetEveryHourToday () {
    List<Tweet> list = new ArrayList<Tweet>();
    Date date = new Date();
    for (int i=0; i < 24; i++) {
        date.setHours(i);
        list.add(new Tweet("rbmllr", "keep it up! you can do it", date));
    }
    return list;
}
```

This code intends to advance a single Date object through the 24 hours of a day, creating a tweet for every hour. But notice that the constructor of Tweet saves the reference that was passed in, so all 24 Tweet objects end up aliasing to the same date:

![Diagram](image.png)

Again, the immutability of Tweet has been violated. We can fix this problem too by judicious defensive copying, this time in the constructor:

```java
public Tweet(String author, String text, Date timestamp) {
    this.author = author;
    this.text = text;
    this.timestamp = new Date(timestamp.getTime());
}
```

In general, you should carefully inspect the argument types and return types of all your ADT operations. If any of the types are mutable, make sure your implementation doesn’t return direct references to its representation.
You may object that this seems wasteful. Why make all these copies of dates? Why can’t we just solve this problem by careful specification:

```java
/**
   * Make a Tweet.
   * @param author    Twitter user who wrote the tweet.
   * @param text      text of the tweet
   * @param timestamp date/time when the tweet was sent. Caller must never mutate this Date object again!
   */
   public Tweet(String author, String text, Date timestamp) {
```

This approach is sometimes taken when there isn’t any other reasonable alternative – for example, when the mutable object is too large to copy efficiently. But the cost in your ability to reason about the program, and your ability to avoid bugs, is enormous. In the absence of compelling arguments to the contrary, it’s almost always worth it for an abstract data type to guarantee its own invariants, and preventing rep exposure is essential to that.

An even better solution is to prefer immutable types. If we had used an immutable date object instead of the mutable `Date`, then we would have ended this section after talking about `public` and `private`. No rep exposure would have been possible.

### Immutable Wrappers Around Mutable Data Types

The Java Collections classes offer an interesting compromise: *immutable wrappers*. `Collections.unmodifiableList()` takes a (mutable) `List` and wraps it with an object that looks like a `List`, but whose mutators are disabled – `set()`, `add()`, `remove()` throw exceptions. So you can construct a list using mutators, then seal it up in an unmodifiable wrapper (and throw away your reference to the original mutable list), and get an immutable list.

The downside here is that you get immutability at runtime, but not at compile time. Java won’t warn you at compile time if you try to `sort()` this unmodifiable list. You’ll just get an exception at runtime. But that’s still better than nothing, so using unmodifiable lists, maps, and sets can be a very good way to reduce the risk of bugs.

### How to Establish Invariants

An invariant is a property that is true for the entire program – which in the case of an invariant about an object, reduces to the entire lifetime of the object.

If the object is a state machine, then we need to:

- establish invariant in the initial state
- ensure that all state transitions preserve the invariant

So your creators and producers must establish the invariant for new instances, and all mutators (and observers, too, but particularly mutators) must preserve it.

Immutable types are simpler, because they have only one state to reason about.

The risk of rep exposure makes the situation more complicated. So the full rule for proving invariants is:

**Structural induction:** If an invariant of an abstract data type is

(1) established by creators;
(2) preserved by producers, mutators, and observers; and (3) no representation exposure occurs, then the invariant is true of all instances of the abstract data type.

**Summary**

Abstract data types are characterized by their operations. Representation independence makes it possible to change the representation of a type without its clients being changed. An abstract data type that preserves its own invariants is easier and safer to use. Java language mechanisms like access control help ensure rep independence and invariants, but representation exposure is a trickier issue, and needs be handled by careful programmer discipline.