L9: Abstract Data Types Part 1

Today
- Abstract data types
- Representation independence
- Representation exposure

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

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright held by the authors, all rights reserved.
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:

\[
\text{creator: } t^* \rightarrow T \\
\text{producer: } T+, t^* \rightarrow T \\
\text{mutator: } T+, t^* \rightarrow \text{void} \\
\text{observer: } T+, t^* \rightarrow t
\]
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 `concat` takes two strings. The occurrences of $t$ on the left may also be omitted; some observers take no non-abstract arguments (e.g., `size`), and some take several.

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

- **int** is Java's primitive integer type. 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)

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

- **String** is Java's string type. String is immutable.
  - creators: `String()`, `String(char[])` constructors
  - producers: `concat()`, `substring()`, `toUpperCase()`
  - observers: `length()`, `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 `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 a few, simple operations that can be combined in powerful ways than lots of complex operations.

- Each operation should have a well-defined purpose, and should have a coherent behavior rather than a panoply of special cases. We probably shouldn't add a `sum` operation to `List`, for example. It might help clients who work with lists of `Integers`, but what about lists of `Strings`? Or nested lists? All these special cases would make `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.

**Preserving Invariants**

Finally, and perhaps most important, a good abstract data type should preserve 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
public class Transaction {
    public int amount;
    public Calendar date;

    public Transaction(int amount, Calendar date) {
        this.amount = amount;
        this.date = date;
    }
}
```

How do we guarantee that `Transaction` objects are immutable — that, once a transaction is created, its date and amount 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
Transaction t = new Transaction(10, new GregorianCalendar());
t.amount += 10;
```

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 `Transaction` 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 Transaction {
    private final int amount;
    private final Calendar date;

    public Transaction(int amount, Calendar date) {
        this.amount = amount;
        this.date = date;
    }

    public int getAmount() {
        return amount;
    }

    public Calendar getDate() {
        return date;
    }
}
```

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 `Transaction`:

```java
/** @return a transaction of same amount as t, one month later */
public static Transaction makeNextPayment(Transaction t) {
    Calendar d = t.getDate();
    d.add(Calendar.MONTH, 1);
    return new Transaction(t.getAmount(), d);
}
```

`makeNextPayment` takes a transaction and should return another transaction for the same amount but dated a month later. The `makeNextPayment` method might be part of a system that schedules recurring payments.
What’s the problem here? The `getDate` call returns a reference to the same calendar object referenced by transaction `t`. So when the calendar object is mutated by `add()`, this affects the date in `t` as well:

![Diagram showing transaction's date and calendar object]

Transaction’s immutability invariant has been broken. The problem is that Transaction leaked out a reference to a mutable object that its invariant depended on. We exposed the rep, in such a way that Transaction 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 Calendar getDate() {
    return (Calendar) date.clone();
}
```

`clone()` is probably the best way to do this with Calendar (despite the unfortunate problems with `clone()` in general – see Josh Bloch, *Effective Java*, item 10). Other classes offer a copy constructor, like `StringBuilder(String)`.

But we’re not done yet! There’s still rep exposure. Consider this (again perfectly reasonable) client code:

```java
/** @return a list of 12 monthly payments of identical amounts */
public static List<Transaction> makeYearOfPayments (int amount) {
    List<Transaction> list = new ArrayList<Transaction>();
    Calendar date = new GregorianCalendar();
    for (int i=0; i < 12; i++) {
        list.add(new Transaction(amount, date));
        date.add(Calendar.MONTH, 1);
    }
    return list;
}
```
This code intends to advance a single Calendar object through 12 months, creating a transaction for each date. But notice that the constructor of Transaction saves the reference that was passed in, so all 12 Transaction objects end up pointing to the same date:

```
public Transaction(int amount, Calendar date) {
    this.amount = amount;
    this.date = (Calendar)date.clone();
}
```

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:

```
/**
 * ...
 * @param date Date of transaction. Caller must never mutate date again!
 */
public Transaction(int amount, Calendar date) { ...}
```

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 Calendar, then we would have ended this section after talking about public and private. No rep exposure would have been possible.

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.

**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.

**Example: String with maxChar Operation**

The Java `final` keyword asks the compiler to stop us from modifying a field after object creation. Unfortunately, when a field has a class type that is not itself immutable, it is still possible to introduce mutations unintentionally, as the rep exposure examples above show. We can even make mistakes with code within the class, not just client code.

However, mutating the data stored inside an “immutable” object need not violate the immutability invariant, thanks to the magic of encapsulation and representation invariance! We just need to be very careful in convincing ourselves that our mutations are consistent with the invariant.

One broad class of examples is “internal” use of mutable state as a performance optimization. 6.005 isn’t about maximizing performance, but it’s important to understand how such activities connect to our core goals in the class.

For a concrete example, imagine we are implementing the functionality of the `String` class ourselves. The class’ sole field might be a character array:

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

We do a good job of following the rules of immutability, never writing to this array after it is initialized. One operation we might want is `maxChar`, which computes which character appearing in the string comes latest in alphabetical order (actually ASCII ordering).
/** Return the character in the string with the highest ASCII code. 
 * @return that character, or 0 for an empty string 
 */

public char maxChar() {
    char ch = 0;
    for (int i = 0; i < a.length; ++i) {
        if (a[i] > ch)
            ch = a[i];
    }
    return ch;
}

This method is easy enough to write, but it has the unfortunate property of iterating through every 
character in the string. For large strings, the loop might run for long enough to be a problem. 
Maybe client code calls maxChar repeatedly, so that we could save time by caching the result, reusing it 
for future calls on the same object.

Concretely, we add two new private fields: one possibly saving the correct result for a call to 
maxChar, and another recording whether the answer in the first one is known to be accurate. (It 
won't be accurate before the first call to maxChar on an object.)

// Sneaky private mutable state, to cache the result of a 
// maxChar() computation!
private char maxCharIs;
private boolean maxCharSet;

We can rewrite maxChar to use this cache when possible:

public char maxChar () {
    if (maxCharSet) {
        // We already know the answer, so return it.
        return maxCharIs;
    } else {
        char ch = 0;
        for (int i = 0; i < a.length; ++i) {
            if (a[i] > ch)
                ch = a[i];
        }
        // Record both the answer and the fact that we know it.
        maxCharIs = ch;
        maxCharSet = true;
        return ch;
    }
}

How can we convince ourselves that our clever String class is legitimately immutable, from client 
code's point of view? The technique we just learned for establishing invariants gets the job done. 
We need to be explicit about the invariant:

Invariant for this example: The contents of array a are never modified after object 
initialization, and whenever maxCharSet is true, maxCharIs holds the maximum 
character from a.
With the invariant stated, we walk through our structural induction scheme. Our analysis here will be cursory, since we haven’t bothered presenting the other methods of our String class, but you can carry out this exercise for your real classes in their entirety, and the whole process can even be made mathematically precise (which is beyond the scope of this class).

1. **Established by creators?**

A usual constructor for the class would assign a a fresh array value. Since this is a creator, we are allowed to modify the values in a, so the first part of the invariant is clearly established. More interestingly, we should initialize maxCharSet to false, so that the second part of the invariant is also trivially satisfied, regardless of the value of maxCharIs.

2. **Preserved by producers, mutators, and observers?**

Here we look only at the case of maxChar. Inspection of the code makes it clear that slots within a are read but not written, just on the basis of Java syntax. For the harder part of the invariant, we consider the two cases of the first if. If maxCharSet starts out set, we change neither it nor maxCharIs, so the invariant is preserved. If maxCharSet starts out not set, we set it to true, which brings the obligation of ensuring that maxCharIs gets the correct value, which it does, since we set it to the result of the straightforward code for computing the max character.

3. **No rep exposure occurs?**

This requirement is easy to verify, since all our fields are private, and the only field with a type that supports further mutation is a. An inspection of the code shows that we never return a or store a reference to it anywhere else, so it cannot be leaked.

□

And that completes our (hand-wavy) proof! How does establishing this invariant pay off? Now we know that maxChar is implemented correctly. The invariant justifies returning maxCharIs whenever maxCharSet is true. We have shown this regardless of what other client code appears in the final program, thanks to encapsulation. In other words, we are very ready for change, since changes to other classes can be made by people who don’t even know about our invariant, and it will be maintained anyway!

**Rep Independence and Testing**

We might want a more code-oriented way of testing if a class is vulnerable to rep exposure. At the same time, the idea of representation independence may also seem murky; what does it mean, concretely? Both concerns can be addressed by the exercise we’ll step through now. Code in the wild will seldom do what we do here, but it’s useful as an educational aid, if nothing else.

What is representation independence? One formal notion is contextual equivalence. Basically, two classes 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. (Don’t worry about the details of this terminology. We won’t use it again, though it may be a useful pointer into more advanced literature on theory.)

OK, so we have a useful notion of two classes being indistinguishable by testing. 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.
For a concrete example, consider rational numbers. We'll create two implementations, each with a constructor like so:

```java
/**
 * Make a new RatNum == (n / d).
 * @param n numerator
 * @param d denominator
 * @throws ArithmeticException if d == 0
 */
public RatNum(int n, int d) throws ArithmeticException { … }
```

...and a multiplication method like this:

```java
public RatNum multiply(RatNum n) { … }
```

We'll also include a `toString` method, which is convenient for testing: it lets us convert the internal state of a class into a common format that supports equality testing.

On to our two implementations! The first, `RatNum`, represents a rational number as a numerator and a denominator, each integers, with the invariant that the fraction is in lowest terms. The second, `RatNumPrimes`, does much the same thing, representing integers as their prime factorizations, allowing an implementation of multiplication via canceling factors. From what we know from high school math, the two approaches should be equivalent, and we can convince ourselves about our implementation using testing.

First, we can verify that the two class' constructors are building equivalent values.

```java
// Check that numer/denom renders as result in both classes.
private static void represent(int numer, int denom, String result) {
    RatNum r = new RatNum(numer, denom);
    RatNumPrimes rp = new RatNumPrimes(numer, denom);

    assertEquals(result, r.toString());
    assertEquals(result, rp.toString());
}

@Test
public void testRepresent() {
    represent(5, 1, "5");
    represent(1, 2, "1/2");
    represent(8, 6, "4/3");
    represent(-7, 2, "-7/2");
    represent(-34, 56, "-17/28");
}
```
Next we can carry out a similar exercise for multiplication.

```java
private static void multiply(int numer1, int denom1,
                             int numer2, int denom2, String result) {
    RatNum r1 = new RatNum(numer1, denom1),
    r2 = new RatNum(numer2, denom2);
    RatNumPrimes rp1 = new RatNumPrimes(numer1, denom1),
    rp2 = new RatNumPrimes(numer2, denom2);

    RatNum res = r1.multiply(r2);
    RatNumPrimes resp = rp1.multiply(rp2);

    assertEquals(result, res.toString());
    assertEquals(result, resp.toString());
}

@Test
public void testMultiply() {
    multiply(5, 1, 6, 1, "30");
    multiply(1, 2, 2, 1, "1");
    multiply(4, 3, 56, 129, "224/387");
    multiply(3, 2, 2, 9, "1/3");
    multiply(8, -25, 5, 2, "-4/5");
}
```

All the tests pass, so it seems likely that we really have implemented the same ADT in two different ways, preserving representation independence. Again, this sort of testing doesn’t show up much in the wild, but this is the way you should be thinking in understanding representation independence. If testing like we used here might fail for your class, that might indicate a design flaw.

**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.