

- [Reading 12: Abstract Data Types](#)
- [What Abstraction Means](#)
- [Classifying Types and Operations](#)
- [Designing an Abstract Type](#)
- [Representation Independence](#)
- [Realizing ADT Concepts in Java](#)
- [Testing an Abstract Data Type](#)
- [Summary](#)



Reading 12: Abstract Data Types

Software in 6.005

Safe from bugs

Correct today and correct in the unknown future.

Easy to understand

Communicating clearly with future programmers, including future you.

Ready for change

Designed to accommodate change without rewriting.

Objectives

Today's class introduces two ideas:

- Abstract data types
- Representation independence

In this reading, 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 problem: clients making assumptions about the type's internal 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.

Access Control in Java

You should already have read: [Controlling Access to Members of a Class](#) in the Java Tutorials.

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 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 the original 6.170, the predecessor to 6.005. 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.
- **Observers** take objects of the abstract type and return objects of a different type. The `size` method of `List`, for example, returns an `int`.
- **Mutators** change objects. The `add` method of `List`, for example, mutates a list by adding an element to the end.

We can summarize these distinctions schematically like this (explanation to follow):

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

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. The $+$ marker indicates that the type may occur one or more times in that part of the signature, and the $*$ marker indicates that it occurs zero or more times. $|$ indicates or. For example, a producer may take two values of the abstract type T , like `String.concat()` does:

- `concat : String \times String \rightarrow String`

Some observers take zero arguments of other types t , such as:

- `size : List \rightarrow int`

... and others take several:

- `regionMatches : String \times boolean \times int \times String \times int \times int \rightarrow boolean`

A creator operation is often implemented as a *constructor*, like `new ArrayList()`. But a creator can simply be a static method instead, like `Arrays.asList()`. A creator implemented as a static method is often called a **factory method**. The various `String.valueOf` methods in Java are other examples of creators implemented as factory methods.

Mutators are often signaled by a `void` return type. A method that returns `void` *must* be called for some kind of side-effect, since otherwise it doesn't return anything. But not all mutators return `void`. For example, `Set.add()` returns a `boolean` that indicates whether the set was actually changed. In Java's graphical user interface toolkit, `Component.add()` returns the object itself, so that multiple `add()` calls can be [chained together](#).

Abstract Data Type Examples

Here are some examples of abstract data types, along with some of their operations, grouped by kind.

`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 `+`, `-`, `*`, `/`
- observers: comparison operators `==`, `!=`, `<`, `>`
- mutators: none (it's immutable)

`List` is Java's list type. `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` 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 reserve the term *producer* only for operations that do no mutation.

Designing an Abstract Type

Designing an abstract type involves choosing good operations and determining how they should behave. Here are a few rules of thumb.

It's better to have **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 **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** in the sense that 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. For example, 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

Critically, 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 and postconditions, 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 data type 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:

```
/** MyString represents an immutable sequence of characters. */
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 data type is allowed to know. 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, however, writing test cases that use `assertEquals` directly on `MyString` objects wouldn't work, because we don't have an equality operation defined on `MyString`. We'll talk about how to implement equality carefully in a later reading. For now, the only operations we can perform with `MyStrings` are the ones we've defined above: `valueOf`, `length`, `charAt`, and `substring`. Our tests have to limit themselves to those operations. For example, here's one test for the `valueOf` operation:

```

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 at the end of this reading.

For now, let's look at a simple representation for `MyString`: just an array of characters, exactly the length of the string, with no extra room at the end. Here's how that internal representation would be declared, as an instance variable within the class:

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

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

```

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;
}

```

Question to ponder: 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?

One problem with this implementation is that it's passing up an opportunity for performance improvement. Because this data type is immutable, the `substring` operation doesn't really have to copy characters out into a fresh array. It could just point to the original `MyString` object's character array and keep track of the start and end that the new `substring` object represents. The `String` implementation in some versions of Java do this.

To implement this optimization, we could change the internal representation of this class to:

```

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) {
    MyString that = new MyString();
    that.a = this.a;
    that.start = this.start + start;
    that.end = this.start + end;
    return that;
}

```

Because `MyString`’s existing clients depend only on the specs of its public methods, not on its private fields, we can make this change without having to inspect and change all that client code. That’s the power of representation independence.

Realizing ADT Concepts in Java

Let’s summarize some of the general ideas we’ve discussed in this reading, which are applicable in general to programming in any language, and their specific realization using Java language features. The point is that there are several ways to do it, and it’s important to both understand the big idea, like a creator operation, and different ways to achieve that idea in practice.

The only item in this table that hasn’t yet been discussed in this reading is the use of a constant object as a creator operation. This pattern is commonly seen in immutable types, where the simplest or emptiest value of the type is simply a public constant, and producers are used to build up more complex values from it.

| ADT concept | Ways to do it in Java | Examples |
|--------------------|-------------------------|-------------------------------------------------------------------------------|
| Creator operation | Constructor | ArrayList() |
| | Static (factory) method | Collections.singletonList() , Arrays.asList() |
| | Constant | BigInteger.ZERO |
| Observer operation | Instance method | List.get() |
| | Static method | Collections.max() |
| Producer operation | Instance method | String.trim() |
| | Static method | Collections.unmodifiableList() |
| Mutator operation | Instance method | List.add() |
| | Static method | Collections.copy() |
| Representation | private fields | |

Testing an Abstract Data Type

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

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('e', s.charAt(3));
}

@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));
}

@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));
}

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

@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));
}

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

```

Notice that each test case typically calls a few operations that *make* or *modify* objects of the type (creators, producers, mutators) and some operations that *inspect* objects of the type (observers). As a result, each test case covers parts of several operations.

Summary

- Abstract data types are characterized by their operations.
- Operations can be classified into creators, producers, observers, and mutators.
- An ADT's specification is its set of operations and their specs.
- A good ADT is simple, coherent, adequate, and representation-independent.
- An ADT is tested by generating tests for each of its operations, but using the creators, producers, mutators, and observers together in the same tests.

These ideas connect to our three key properties of good software as follows:

- **Safe from bugs.** A good ADT offers a well-defined contract for a data type, so that clients know what to expect from the data type, and implementors have well-defined freedom to vary.
- **Easy to understand.** A good ADT hides its implementation behind a set of simple operations, so that programmers using the ADT only need to understand the operations, not the details of the implementation.
- **Ready for change.** Representation independence allows the implementation of an abstract data type to change without requiring changes from its clients.