Reading 12: Abstract Data Types

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	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 (http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html)** in the Java Tutorials.

READING EXERCISES

The following questions use the code below. Study it first, then answer the questions.

```
    class Wallet {
        private int amount;

        public void loanTo(Wallet that) {
            // put all of this wallet's money into that wallet
/*A*/       that.amount += this.amount;
/*B*/       amount = 0;
        }

        public static void main(String[] args) {
/*C*/       Wallet w = new Wallet();
/*D*/       w.amount = 100;
/*E*/       w.loanTo(w);
        }
    }

    class Person {
        private Wallet w;

        public int getNetWorth() {
/*F*/       return w.amount;
        }

        public boolean isBroke() {
/*G*/       return Wallet.amount == 0;
        }
    }
```

Access control A

Which of the following statements are true about the line marked `/*A*/` ?

```
that.amount += this.amount;
```

✖

- [ ] The reference to `this.amount` is allowed by Java. ☑
- [ ] The reference to `this.amount` is not allowed by Java because it uses `this` to access a private field.
- [x] The reference to `that.amount` is allowed by Java. ☑

- [ ] The reference to `that.amount` is not allowed by Java because `that.amount` is a private field in a different object.
- [ ] The reference to `that.amount` is not allowed by Java because it writes to a private field.
- [ ] The illegal access(es) are caught statically.
- [ ] The illegal access(es) are caught dynamically.

Private fields and methods can be used by any code in the same class. `Wallet`'s private fields and methods can be used by any code in the `Wallet` class, even to access private fields in more than one `Wallet` object, not just `this`. Roughly speaking, any code within the curly braces of `Wallet`'s class body can touch `Wallet`'s private fields and methods. This isn't strictly true, because `Wallet` might contain nested class definitions that don't automatically get access to `Wallet`'s private fields and methods, but aside from that, it's a useful rule of thumb.

CHECK EXPLAIN

Access control B

Which of the following statements are true about the line marked `/*B*/`?

```
amount = 0;
```

- [ ] The reference to `amount` is allowed by Java.
- [ ] The reference to `amount` is not allowed by Java because it doesn't use `this`.
- [ ] The illegal access is caught statically.
- [x] The illegal access is caught dynamically.

Private fields and methods can be used by any code in the same class. For fields, the `this` reference is implicit and can be omitted.

CHECK EXPLAIN

Access control C

Which of the following statements are true about the line marked `/*C*/`?

```
Wallet w = new Wallet();
```

- [ ] The call to the `Wallet()` constructor is allowed by Java.
- [ ] The call to the `Wallet()` constructor is not allowed by Java because there is no public `Wallet()` constructor declared.
- [x] The illegal access is caught statically.
- [ ] The illegal access is caught dynamically.

In Java, a class with no explicitly-declared constructors gets an automatic public constructor that takes no arguments.

CHECK EXPLAIN

Access control D

Which of the following statements are true about the line marked `/*D*/` ?

```
w.amount = 100;
```

- [ ] The access to `w.amount` is allowed by Java.
- [x] The access to `w.amount` is not allowed by Java because amount is private.
- [ ] The illegal access is caught statically.
- [ ] The illegal access is caught dynamically.

`Wallet` 's private fields and methods can be used by any code in the `Wallet` class, even static methods. Roughly speaking, any code within the curly braces of `Wallet` 's class body can touch `Wallet` 's private fields and methods. This isn't strictly true because `Wallet` might contain nested class definitions that don't automatically get access to `Wallet` 's private fields and methods, but aside from that, it's a useful rule of thumb.

CHECK EXPLAIN

Access control E

Which of the following statements are true about the line marked `/*E*/`

```
w.loanTo(w);
```

- [x] The call to `loanTo()` is allowed by Java.
- [ ] The call to `loanTo()` is not allowed by Java because `this` and `that` will be aliases to the same object.
- [ ] The problem will be found by a static check.
- [ ] The problem will be found by a dynamic check.
- [x] After this line, the `Wallet` object pointed to by `w` will have amount 0.
- [ ] After this line, the `Wallet` object pointed to by `w` will have amount 100.
- [ ] After this line, the `Wallet` object pointed to by `w` will have amount 200.

In this call to `loanTo()`, `this` and `that` will indeed be aliases for the same object, but Java doesn't prevent it. It causes `loanTo()` to behave badly, emptying out the wallet.

CHECK EXPLAIN

Access control F

Which of the following statements are true about the line marked `/*F*/` ?

```
return w.amount;
```

- [ ] The reference to `w.amount` is allowed by Java because both `w` and `amount` are private variables.
- [x] The reference to `w.amount` is allowed by Java because `amount` is a primitive type, even though it's private.
- [ ] The reference to `w.amount` is not allowed by Java because `amount` is a private field in a different class.
- [x] The illegal access is caught statically.
- [ ] The illegal access is caught dynamically.

This code is not in the `Wallet` class, so it is not allowed to access `Wallet`'s private fields and methods. Java produces a static compiler error about it.

CHECK EXPLAIN

Access control G

Which of the following statements are true about the line marked `/*G*/`?

```
return Wallet.amount == 0;
```

- [ ] The reference to `Wallet.amount` is allowed by Java because `Wallet` has permission to access its own private field `amount`.
- [ ] The reference to `Wallet.amount` is allowed by Java because `amount` is a static variable.
- [x] The reference to `Wallet.amount` is not allowed by Java because `amount` is a private field.
- [ ] The reference to `Wallet.amount` is not allowed by Java because `amount` is an instance variable.
- [x] The illegal access is caught statically.
- [ ] The illegal access is caught dynamically.

`amount` is an instance variable, and requires a specific `Wallet` object instance to access. So you can't refer to it with the class name `Wallet` like you would a static variable, you have to use an instance object like `w` in the previous examples.

`amount` is also private, so not accessible here.

Both problems would have to be fixed – changing `amount` to `public static` – in order to make the code legal. But doing this would certainly be a bad idea, because it would make `amount` a global variable. Everybody would be able to look at and change your wallet. In fact there would only be a single `amount` variable, shared by everybody's wallet!

CHECK EXPLAIN

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* → T
- producer: T+, t* → T
- observer: T+, t* → t
- mutator: T+, t* → void|t|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. For example, a producer may take two values of the abstract type, like `String.concat()` does. The occurrences of t on the left may also be omitted, since some observers take no non-abstract arguments, and some take several.

A creator operation is often implemented as a *constructor*, like `new ArrayList()` (https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList--). But a creator can simply be a static method instead, like `Arrays.asList()` (http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-). A creator implemented as a static method is often called a **factory method**. The various `String.valueOf` (http://docs.oracle.com/javase/8/docs/api/java/lang/String.html#valueOf-boolean-) 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() (http://docs.oracle.com/javase/8/docs/api/java/util/Set.html#add-E-) returns a boolean that indicates whether the set was actually changed. In Java's graphical user interface toolkit, Component.add() (http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#add-java.awt.Component-) returns the object itself, so that multiple add() calls can be chained together (http://en.wikipedia.org/wiki/Method_chaining).

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` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T-)
- producers: `Collections.unmodifiableList` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList-T-)
- observers: `size`, `get`
- mutators: `add`, `remove`, `addAll`, `Collections.sort` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#sort-java.util.List-)

`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.

READING EXERCISES

Operations

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.

READING EXERCISES

Representation 1

Consider the following abstract data type.

```
/**
 * Represents a family that lives in a household together.
 * A family always has at least one person in it.
 * Families are mutable.
 */
class Family {
    // the people in the family, sorted from oldest to youngest, with no dupli
cates.
    public List<Person> people;

    /**
     * @return a list containing all the members of the family, with no duplic
ates.
     */
    public List<Person> getMembers() {
        return people;
    }
}
```

Here is a client of this abstract data type:

```
void client1(Family f) {
    // get youngest person in the family
    Person baby = f.people.get(f.people.size()-1);
    ...
}
```

Assume all this code works correctly (both `Family` and `client1`) and passes all its tests.

Now `Family`'s representation is changed from a `List` to `Set`, as shown:

```
/**
 * Represents a family that lives in a household together.
 * A family always has at least one person in it.
 * Families are mutable.
 */
class Family {
    // the people in the family
    public Set<Person> people;

    /**
     * @return a list containing all the members of the family, with no duplic
ates.
     */
    public List<Person> getMembers() {
        return new ArrayList<Person>(people);
    }
}
```

Assume that `Family` compiles correctly after the change.

Which of the following statements are true about `client1` after `Family` is changed?

- ○ `client1` is independent of `Family`'s representation, so it keeps working correctly.
- ◉ `client1` depends on `Family`'s representation, and the dependency would be caught as a static error.
- ○ `client1` depends on `Family`'s representation, and the dependency would be caught as a dynamic error.
- ○ `client1` depends on `Family`'s representation, and the dependency would not be caught but would produce a wrong answer at runtime.
- ○ `client1` depends on `Family`'s representation, and the dependency would not be caught but would (luckily) still produce the same answer.

`client1` is directly accessing the `people` field in `Family`. When that field was a `List`, it could call `get()` on it with no trouble. Now that the field is a `Set`, the `get()` method doesn't exist, so there is a static error.

Because `client1` no longer works when `Family`'s representation changes, we say that `client1` has a dependency on `Family`'s representation.

CHECK EXPLAIN

Representation 2

Now consider `client2`:

```
void client2(Family f) {
    // get size of the family
    int familySize = f.people.size();
    ...
}
```

Which of the following statements are true about client2 after Family is changed?

✖

- ○ `client2` is independent of `Family`'s representation, so it keeps working correctly.
- ○ `client2` depends on `Family`'s representation, and the dependency would be caught as a static error.
- ◉ `client2` depends on `Family`'s representation, and the dependency would be caught as a dynamic error.
- ○ `client2` depends on `Family`'s representation, and the dependency would not be caught but would produce a wrong answer at runtime.
- ○ `client2` depends on `Family`'s representation, and the dependency would not be caught but would (luckily) still produce the same answer.

☑

›

`client2` is also directly accessing the people field in `Family`. Both the `List` version and the `Set` version of that field have a `size()` method, and since the `List` had no duplicates, `size()` in both cases returns the correct size of the family.

But we still say that `client2` has a dependency on `Family`'s representation; it just got lucky this time. If the type of `people` field had instead changed to `People[]`, then `client2` would no longer work, because it needs to use `.length` instead of `size()`.

CHECK EXPLAIN

Representation 3

Now consider `client3`:

```
void client3(Family f) {
    // get any person in the family
    Person anybody = f.getMembers().get(0);
    ...
}
```

Which of the following statements are true about `client3` after `Family` is changed?

✖

- ○ `client3` is independent of `Family`'s representation, so it keeps working correctly. ☑
- ○ `client3` depends on `Family`'s representation, and the dependency would be caught as a static error.
- ○ `client3` depends on `Family`'s representation, and the dependency would be caught as a dynamic error.
- ○ `client3` depends on `Family`'s representation, and the dependency would not be caught but would produce a wrong answer at runtime.
- ◉ `client3` depends on `Family`'s representation, and the dependency would not be caught but would (luckily) still produce the same answer.

›

`client3` is calling a public method to get the members of the family, and it depends only on the contract of that public method. Note that the contract of `getMembers()` doesn't say anything about the order of the people in the list it returns, but `client3` doesn't care about that ordering anyway. `client3` only cares that the list has at least one person in it, and `Family`'s contract as a whole promises that.

Since `getMembers()` still satisifes its contract, even with the new `Set` representation, `client3` will keep working after the change. It is independent of the representation.

CHECK EXPLAIN

Representation 4

For each section of the Family data type's code shown below, is it part of the ADT's specification, its representation, or its implementation?

✔ specification — specification

```
/**
 * Represents a family that lives in a household toget
her.
 * A family always has at least one person in it.
 * Families are mutable.
 */
```

✔ specification — specification

```
public class Family {
```

✔ representation — representation

```
    // the people in the family, sorted from oldest to
youngest, with no duplicates.
```

✖ implementation — representation

```
    private List<Person> people;
```

✖ implementation — specification

```
    /**
     * @return a list containing all the members of th
e family, with no duplicates.
     */
```

✖ representation — specification

```
    public List<Person> getMembers() {
```

✖ specification — implementation

```
        return people;
```

```
    }
}
```

The specification of an ADT includes the name of the class (on line 6), the Javadoc comment just before the class (lines 1-5), and the specifications of its public methods and fields (Javadoc lines 10-12 and method signature line 13). These parts are the contract that is visible to a client of the class.

The representation of an ADT consists of its fields (line 8) and any assumptions or requirements about those fields (line 7).

The implementation of an ADT consists of the method implementations that manipulate its rep (line 14).

CHECK EXPLAIN

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()` (http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList--)
	Static (factory) method	`Collections.singletonList()` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T-), `Arrays.toList()` (http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-)
	Constant	`BigInteger.ZERO` (http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#ZERO)
Observer operation	Instance method	`List.get()` (http://docs.oracle.com/javase/8/docs/api/java/util/List.html#get-int-)

		Static method	`Collections.max()` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#max-java.util.Collection-)
	Producer operation	Instance method	`String.trim()` (http://docs.oracle.com/javase/8/docs/api/java/lang/String.html#trim--)
		Static method	`Collections.unmodifiableList()` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList-java.util.List-)
	Mutator operation	Instance method	`List.add()` (http://docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E-)
		Static method	`Collections.copy()` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#copy-java.util.List-java.util.List-)
	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.

READING EXERCISES

These questions use the following datatype:

```
/** Immutable datatype representing a student's progress through school. */
class Student {

    /** make a freshman */
    public Student() { ... }


    /** @return a student promoted to the next year, i.e.
           freshman returns a sophomore,
           sophomore returns a junior,
           junior returns a senior,
           senior returns an alum,
           alum stays an alum and can't be promoted further. */
    public Student promote() { ... }


    /** @return number of years of school completed, i.e.
           0 for a freshman, 4 for an alum */
    public int getYears() { ... }

}
```

Partitioning ADT operations

How many parts are there in a reasonable input-space partition of the `Student()` constructor?

✖ 0

How many parts are there in a reasonable, but not exhaustive, input-space partition of `promote()`?

✖ 1

How many parts are there in a reasonable, but not exhaustive, input-space partition of `getYears()`?

✖ 1

CHECK EXPLAIN

Choosing ADT test cases

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.