Reading 14: Interfaces

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

The topic of today's class is interfaces: separating the interface of an abstract data type from its implementation, and using Java `interface` types to enforce that separation.

After today's class, you should be able to define ADTs with interfaces, and write classes that implement interfaces.

Interfaces

Java's `interface` is a useful language mechanism for expressing an abstract data type. An interface in Java is a list of method signatures, but no method bodies. A class *implements* an interface if it declares the interface in its `implements` clause, and provides method bodies for all of the interface's methods. So one way to define an abstract data type in Java is as an interface, with its implementation as a class implementing that interface.

One advantage of this approach is that the interface specifies the contract for the client and nothing more. The interface is all a client programmer needs to read to understand the ADT. The client can't create inadvertent dependencies on the ADT's rep, because instance variables can't be put in an interface at all. The implementation is kept well and truly separated, in a different class altogether.

Another advantage is that multiple different representations of the abstract data type can co-exist in the same program, as different classes implementing the interface. When an abstract data type is represented just as a single class, without an interface, it's harder to have multiple representations. In the `MyString` example from *Abstract Data Types* , `MyString` was a single class. We explored two different representations for `MyString` , but we couldn't have both representations for the ADT in the same program.

Java's static type checking allows the compiler to catch many mistakes in implementing an ADT's contract. For instance, it is a compile-time error to omit one of the required methods, or to give a method the wrong return type. Unfortunately, the compiler doesn't check for us that the code adheres to the specs of those methods that are written in documentation comments.

> For the details of how to define interfaces in Java, consult the **Java Tutorials section on interfaces** .

Subtypes

Recall that a *type* is a set of values. The Java `List` type is defined by an interface. If we think about all possible `List` values, none of them are `List` objects: we cannot create instances of an interface. Instead, those values are all `ArrayList` objects, or `LinkedList` objects, or objects of another class that implements `List` . A *subtype* is simply a subset of the *supertype* : `ArrayList` and `LinkedList` are subtypes of `List` .

"B is a subtype of A" means "every B is an A." In terms of specifications: "every B satisfies the specification for A."

That means B is only a subtype of A if B's specification is at least as strong as A's specification. When we declare a class that implements an interface, the Java compiler enforces part of this requirement automatically: for example, it ensures that every method in A appears in B, with a compatible type signature. Class B cannot implement interface A without implementing all of the methods declared in A.

But the compiler cannot check that we haven't weakened the specification in other ways: strengthening the precondition on some inputs to a method, weakening a postcondition, weakening a guarantee that the interface abstract type advertises to clients. If you declare a subtype in Java — implementing an interface is our current focus — then you must ensure that the subtype's spec is at least as strong as the supertype's.

Example: `MyString`

Let's revisit `MyString` . Using an interface instead of a class for the ADT, we can support multiple implementations:

```
/** MyString represents an immutable sequence of characters. */
public interface MyString {

    // We'll skip this creator operation for now
    // /** @param b a boolean value
    //  *  @return string representation of b, either "true" or "false" */
    // public static MyString valueOf(boolean b) { ... }

    /** @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);

    /** 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);
}
```

We'll skip the static `valueOf` method and come back to it in a minute. Instead, let's go ahead using a different technique from our toolbox of ADT concepts in Java : constructors.

Here's our first implementation:

```
public class SimpleMyString implements MyString {

    private char[] a;

    /* Create an uninitialized SimpleMyString. */
    private SimpleMyString() {}

    /** Create a string representation of b, either "true" or "false".
     *  @param b a boolean value */
    public SimpleMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
              : new char[] { 'f', 'a', 'l', 's', 'e' };
    }

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

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

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

And here's the optimized implementation:

```
public class FastMyString implements MyString {

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

    /* Create an uninitialized FastMyString. */
    private FastMyString() {}

    /** Create a string representation of b, either "true" or "false".
     *  @param b a boolean value */
    public FastMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
              : new char[] { 'f', 'a', 'l', 's', 'e' };
        start = 0;
        end = a.length;
    }

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

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

    @Override public MyString substring(int start, int end) {
        FastMyString that = new FastMyString();
        that.a = this.a;
        that.start = this.start + start;
        that.end = this.start + end;
        return that;
    }
}
```

- Compare these classes to the implementations of `MyString` in *Abstract Data Types* . Notice how the code that previously appeared in static `valueOf` methods now appears in the constructors, slightly changed to refer to the rep of `this` .
- Also notice the use of `@Override` . This annotation informs the compiler that the method must have the same signature as one of the methods in the interface we're implementing. But since the compiler already checks that we've implemented all of the interface methods, the primary value of `@Override` here is for readers of the code: it tells us to look for the spec of that method in the interface. Repeating the spec wouldn't be DRY, but saying nothing at all makes the code harder to understand.
- And notice the private empty constructors we use to make new instances in `substring(..)` before we fill in their reps with data. We didn't have to write these empty constructors before because Java provides them by default when we don't declare any others. Adding the constructors that take `boolean b` means we have to declare the empty constructors explicitly.

  Now that we know good ADTs scrupulously preserve their own invariants , these do-nothing constructors are a **bad** pattern: they don't assign any values to the rep, and they certainly don't establish any invariants. We should strongly consider revising the implementation. Since `MyString` is immutable, a starting point would be making all the fields `final` .

How will clients use this ADT? Here's an example:

```
MyString s = new FastMyString(true);
System.out.println("The first character is: " + s.charAt(0));
```

This code looks very similar to the code we write to use the Java collections classes:

```
List<String> s = new ArrayList<String>();
...
```

Unfortunately, this pattern **breaks the abstraction barrier** we've worked so hard to build between the abstract type and its concrete representations. Clients must know the name of the concrete representation class. Because interfaces in Java cannot contain constructors, they must directly call one of the concrete class' constructors. The spec of that constructor won't appear anywhere in the interface, so there's no static guarantee that different implementations will even provide the same constructors.

Fortunately, (as of Java 8) interfaces *are* allowed to contain static methods, so we can implement the creator operation `valueOf` as a static factory method in the interface `MyString` :

```
public interface MyString {

    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) {
        return new FastMyString(true);
    }

    // ...
```

Now a client can use the ADT without breaking the abstraction barrier:

```
MyString s = MyString.valueOf(true);
System.out.println("The first character is: " + s.charAt(0));
```

Example: `Set`

Java's collection classes provide a good example of the idea of separating interface and implementation.

Let's consider as an example one of the ADTs from the Java collections library, `Set`. `Set` is the ADT of finite sets of elements of some other type `E`. Here is a simplified version of the `Set` interface:

```
/** A mutable set.
 *  @param <E> type of elements in the set */
public interface Set<E> {
```

`Set` is an example of a *generic type*: a type whose specification is in terms of a placeholder type to be filled in later. Instead of writing separate specifications and implementations for `Set<String>`, `Set<Integer>`, and so on, we design and implement one `Set<E>`.

We can match Java interfaces with our classification of ADT operations, starting with a creator:

```
    // example creator operation
    /** Make an empty set.
     *  @param <E> type of elements in the set
     *  @return a new set instance, initially empty */
    public static <E> Set<E> make() { ... }
```

The `make` operation is implemented as a static factory method. Clients will write code like:
`Set<String> strings = Set.make();`
and the compiler will understand that the new `Set` is a set of `String` objects.

```
    // example observer operations
    /** Get size of the set.
     *  @return the number of elements in this set */
    public int size();

    /** Test for membership.
     *  @param e an element
     *  @return true iff this set contains e */
    public boolean contains(E e);
```

Next we have two observer methods. Notice how the specs are in terms of our abstract notion of a set; it would be malformed to mention the details of any particular implementation of sets with particular private fields. These specs should apply to any valid implementation of the set ADT.

```
    // example mutator operations

    /** Modifies this set by adding e to the set.
     *  @param e element to add */
    public void add(E e);

    /** Modifies this set by removing e, if found.
     *  If e is not found in the set, has no effect.
     *  @param e element to remove */
    public void remove(E e);
```

The story for these mutators is basically the same as for the observers. We still write specs at the level of our abstract model of sets.

In the Java Tutorials, read these pages:

- **Lesson: Interfaces**
- **The Set Interface**
- **Set Implementations**
- **The List Interface**
- **List Implementations**

Generic Interfaces

Suppose we want to implement the generic `Set<E>` interface above.

Generic interface, non-generic implementation. One way we might do this is to implement `Set<E>` for a *particular* type `E`.

In *Abstraction Functions & Rep Invariants* we looked at `CharSet`, which represents a set of characters. The example code for `CharSet` includes a generic `Set` interface and each of the implementations `CharSet1` / `2` / `3` declare:

```
public class CharSet implements Set<Character>
```

When the interface mentions placeholder type `E`, the `CharSet` implementations replace `E` with `Character`. For example:

```
public interface Set<E> {

    // ...

    /**
     * Test for membership.
     * @param e an element
     * @return true iff this set contains e
     */
    public boolean contains(E e);

    /**
     * Modifies this set by adding e to the set.
     * @param e element to add
     */
    public void add(E e);

    // ...
}
```

```
public class CharSet1 implements Set<Character> {

    private String s = "";


    // ...


    @Override
    public boolean contains(Character e) {
        checkRep();
        return s.indexOf(e) != -1;
    }

    @Override
    public void add(Character e) {
        if (!contains(e)) s += e;
        checkRep();
    }
    // ...
}
```

The representations used by `CharSet1` / `2` / `3` are not suited for representing sets of arbitrary-type elements. The `String` reps, for example, cannot represent a `Set<Integer>` without careful work to define a new rep invariant and abstraction function that handles multi-digit numbers.

Generic interface, generic implementation. We can also implement the generic `Set<E>` interface without picking a type for `E`. In that case, we write our code blind to the actual type that clients will choose for `E`. Java's `HashSet` does that for `Set`. It's declaration looks like:

```
public interface Set<E> {

    // ...
```

```
public class HashSet<E> implements Set<E> {

    // ...
```

A generic implementation can only rely on details of the placeholder types that are included in the interface's specification. We'll see in a future reading how `HashSet` relies on methods that every type in Java is required to implement — and only on those methods, because it can't rely on methods declared in any specific type.

Why Interfaces?

Interfaces are used pervasively in real Java code. Not every class is associated with an interface, but there are a few good reasons to bring an interface into the picture.

- **Documentation for both the compiler and for humans** . Not only does an interface help the compiler catch ADT implementation bugs, but it is also much more useful for a human to read than the code for a concrete implementation. Such an implementation intersperses ADT-level types and specs with implementation details.
- **Allowing performance trade-offs** . Different implementations of the ADT can provide methods with very different performance characteristics. Different applications may work better with different choices, but we would like to code these applications in a way that is representation-independent. From a correctness standpoint, it should be possible to drop in any new implementation of a key ADT with simple, localized code changes.
- **Optional methods** . `List` from the Java standard library marks all mutator methods as optional. By building an implementation that does not support these methods, we can provide immutable lists. Some operations are hard to implement with good enough performance on immutable lists, so we want mutable implementations, too. Code that doesn't call mutators can be written to work automatically with either kind of list.
- **Methods with intentionally underdetermined specifications** . An ADT for finite sets could leave unspecified the element order one gets when converting to a list. Some implementations might use slower method implementations that manage to keep the set representation in some sorted order, allowing quick conversion to a sorted list. Other implementations might make many methods faster by not bothering to support conversion to sorted lists.
- **Multiple views of one class** . A Java class may implement multiple methods. For instance, a user interface widget displaying a drop-down list is natural to view as both a widget and a list. The class for this widget could implement both interfaces. In other words, we don't implement an ADT multiple times just because we are choosing different data structures; we may make multiple implementations because many different sorts of objects may also be seen as special cases of the ADT, among other useful perspectives.
- **More and less trustworthy implementations** . Another reason to implement an interface multiple times might be that it is easy to build a simple implementation that you believe is correct, while you can work harder to build a fancier version that is more likely to contain bugs. You can choose implementations for applications based on how bad it would be to get bitten by a bug.

Realizing ADT Concepts in Java

We can now extend our Java toolbox of ADT concepts from the first ADTs reading:

ADT concept	Ways to do it in Java	Examples
Abstract data type	Single class	`String`
	Interface + class(es)	`List` and `ArrayList`
Creator operation	Constructor	`ArrayList()`
	Static (factory) method	`Collections.singletonList()` , `Arrays.toList()`
	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	

Summary

Java interfaces help us formalize the idea of an abstract data type as a set of operations that must be supported by a type.

This helps make our code…

- **Safe from bugs.** An ADT is defined by its operations, and interfaces do just that. When clients use an interface type, static checking ensures that they only use methods defined by the interface. If the implementation class exposes other methods — or worse, has visible representation — the client can't accidentally see or depend on them. When we have multiple implementations of a data type, interfaces provide static checking of the method signatures.
- **Easy to understand.** Clients and maintainers know exactly where to look for the specification of the ADT. Since the interface doesn't contain instance fields or implementations of instance methods, it's easier to keep details of the implementation out of the specifications.
- **Ready for change.** We can easily add new implementations of a type by adding classes that implement interface. If we avoid constructors in favor of static factory methods, clients will only see the interface. That means we can switch which implementation class clients are using without changing their code at all.