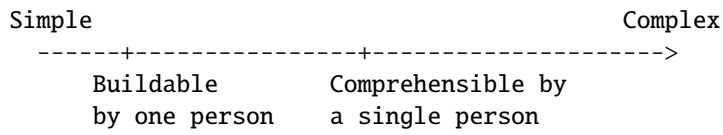
Lecture 4: ADTs, Interfaces, and Generics

built on 2019/05/01 at 10:52:05

How do you express what a class must do, separately from how it will do it? A great motivating factor is when you develop a large software system.



As the project size grows larger, you need proper division of labor. But....

How many programmer A use (future) code of programmer B?

1 Recap #1: Java Interface

More concretely: you tell your friend, write a class A for me. How do you make progress on your own implementation that uses A, even before A is completed?

You need a way to say A will have methods x, y, and z, etc. etc. This "contract" (will be called *interface*) allows you to use any implementation as long as it satisfies the contract (aka. it implements the interface).

1.1 Introducing Java Interface

Quite similar to a class, you define it with the **`interface`** keyword (often **`public interface`** to make it publicly visible). Inside it, you specify

1. methods that are needed to satisfy the contract, as well as
2. variables that are part of the contract.

where

- Methods: implicitly public
- Variables: implicitly public, static, and final — and must be initialized.

As an example, we'll write an interface to a class that generates a series of numbers:

```
public interface Series {
  int next();  // return next number in series
  void reset();  // restart
  void setStart(int x);  // set starting value
}
```

This creates an interface that promises 3 methods.

Implementing Interfaces: To implement it, we'll define a class and indicate that the class implements this interface, like so:

```
public class ByTwos implements Series {
  private int start;
  private int val;
  ByTwos() {
    start = 0;
    val = 0;
  }
```

```
  public int next() {
    int prevVal = val;
    val += 2;
    return prevVal;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}
```

We can still use this class as usual:

```
public class SeriesDemo {
  public static void main(String args[]) {
    ByTwos seq = new ByTwos();
    for(int i=0; i < 5; i++)
      System.out.println("Next value is " + seq.next());
    System.out.println("\nResetting");
    seq.reset();
    for (int i=0; i < 5; i++)
       System.out.println("Next value is " + seq.next());
  }
}
```

NOTE: We can have more methods than specified by the interface. DEMO: `public int foo() { return 42; }`.

Let's now build another class:

```
class ByFives implements Series {
    // instead of adding 2, it is mult-ed by 5 – same as before
}
```

Using Interface References: We'll write code that makes use of the interface.

```
public class SeriesDemo {
  public static void main(String args[]) {
     ByTwos twoSeq = new ByTwos();
     ByFives fiveSeq = new ByFives();

     // Series a = twoSeq; // POLYMORPHISM: twoSeq is a ByTwos – and is like a Series
     // Series b = fiveSeq; // POLYMORPHISM: fiveSeq is a ByFives – and is like a Series

     Series[] sequences = { twoSeq, fiveSeq }; // so this fits in an array of Series

     // Q: Is a.prev() possible here?

     for(int i=0; i < 5; i++) {
```

```
        System.out.printf("Next:");
        for (Series seq : sequences) {
            System.out.printf(" %d", seq.next());
        }
       System.out.printf("\n");
    }
  }
}
```

NOTES:

(1) Variables in Interfaces: Possible, not covered. See the Internet for details.

(2) A class can implement many interfaces.
Syntax: `class A implements I1, I2, I3` This means `class A` satisfies all of `I1`, `I2`, `I3`.

(3) You cannot `new` an interface — you can ascribe (fit into the shape of) an object instance to it.

2 Recap #2: Inheritance

Inheritance lets you acquire the characteristics of a parent class. Consider the following (contrived) example, starting with an `Animal` class:

```
class Animal {
   void walk() { System.out.println("Animal: I'm walking."); }
   void eat()  { System.out.println("Animal: I'm eating.");  }
}
```

The class has two methods walk and eat. Next, we're going to build a `Pet` class, which (mechanically) inherits from `Animal`—that is, it has all the services/properties/characteristics of the `Animal` class and then some.

```
class Pet extends Animal {
   void play(String with) { System.out.println("Pet: I enjoy playing with " + with); }
}
```

The keyword here is `extends`. In this case, the class `Pet` extends `Animal`. Notice that a Pet can walk, eat, and play with someone.

Multilevel: We don't have to stop here. We can continue the inheritance structure. For example:

```
class Parrot extends Pet {
  void sing() { System.out.println("Parrot: I'm parroting"); }
}
```

Notice that Parrot has all the "abilities" of the parent class (`Pet`), which, in turn, has all the "abilities" of its parent.

Diagramatically: we write....

```
   Animal  (super-)
     ^
     |
    Pet   (sub-)  (super-)
     ^
     |
   Parrot  (sub-)
```

3 Interfaces from a Data Struturing Point of View

We'll discuss the concept of abstract data types (ADTs), a seemingly-simple but great idea where we should shield users from the specifics of that implementation—rather, the users focus on understanding the "contract" made about the contact point (aka. the interface). We will then look at how we can implement this in Java using a combination of classes and interfaces.

3.1 Abstract Data Types

Many software engineers go by the rule of thumb that *no function should ever exceed a page* unless there's a good reason to do so. In many ways, such a principle sounds hard to accomplish. How can one write a working program in less than a page? Like you have seen before, good programs are broken down into pieces, where each piece typically represents a logical unit (more or less a self-contained unit that carries out certain logic). These pieces call other pieces to accomplish a task.

This very idea—often called modular design—has several benefits:

- Ease writing/debugging code. Small things are easier to write, reason about, and fix when they go wrong. Often, it hides a can of worms from lay programmers.
- Enable working as a team. Different logical units could be developed/tested in parallel in isolation of the other pieces until integration.
- Limit dependencies. For example, this makes it easy to swap out a piece that carries out a certain task for a different piece that does the same task better.

As a working definition for this class, an abstract data type (ADT) **is an interface** that defines a set of operations and the input/output behaviors these functions must have. It, however, **says nothing about how each operation should or must be implemented**. This means there could be many implementations of varying efficiency for the same ADT.

The basic idea here is that there are certain processes that are common across different functions of a program or even across programs—and in the perfect world, we want to write code to support such processes once and reuse them wherever they surface. The implementation of an ADT often refers to other ADTs.

Let's look at an example to what this is in action. Our running example asks us to monitor the largest data item in a collection. There's no fixed rule for designing an ADT. Here's one that fits the requirement and will be used in the rest of this lecture. We will create a Max ADT, an ADT supporting:

(1) `new(): void` — create a new instance of this data type (returns nothing);
(2) `add(x: int): void` — add an integer x to the collection; and
(3) `getMax(): int` — retrieve the item that has the largest value.

Let's note that so far we've said nothing about how they will be implemented or how long each of these operations will take. The description might suggest an implementation but doesn't specify one.

3.2 MaxADT

Continuing with the running example, we'll write an interface for our MaxADT. Since this is specific to integer data, we'll call it `MaxIntInterface`. The code is as follows:

```
public interface MaxIntInterface {
  // add elt to the collection
  public void add(int elt);

  // return the maximum value observed so far
  public int getMax();
}
```

The comment above each method declaration is a standard way of reminding the programmers how each method should behave.

4 Satisfying the Contract Using Java Classes

Through a few examples, we'll look at how we can implement ADTs using Java classes. Instead of giving you a lecture on syntax etc., we'll do an integrated example where we'll learn the syntax in context.

4.1 SlowIntMax

Our first example is the problem posed at the start of the lecture: maintaining the maximum value. A moment ago, we have designed an interface/ADT for this problem: the MaxADT data type conceptually (and I'll say more why I say conceptually) maintains a collection of data items and supports *three* operations: `new`(), `add`(x), and `getMax`().

In our initial attempt, we will implement this ADT directly from the description of each operation. Two questions to ask ourselves: (1) what should it maintain? and (2) what does it have to support?

Java uses the following syntax to define a class as well as operations that belong to the class. We might as well define placeholders for all the operations we intend to support.

```
public class SlowMaxInt implements MaxIntInterface {
  public SlowMaxInt() {
    // the recipe for how to create a new instance
  };

  public void add(int elt) {
    // must deposit elt to the collection
  }

  public int getMax() {
    // find and return the maximum value
  }
}
```

An important feature of this class declaration is on the first line: we say that `SlowMaxInt` implements `MaxIntInterface`. This is promising that what we're writing will satisfy the interface given by `MaxIntInterface`.

To complete this implementation, let's sketch a plan. We will:

- Keep a list of numbers (call this `numbers`).
- When the `add` operation is called, add the new number to `numbers`.
- When the `getMax` is called, scan `numbers` list to find the largest item.

We need an ability to remember data/information inside the class to continue. Indeed, a class instance can store data. This is an important feature of classes that distinguish them from ordinary functions.

As a first step, we'll declare an empty `numbers` list upon creating an instance.

> **Q:** Which function is called when we make an instance?
> **A:** The special function called the constructor. It has the same name as the class, in this case `SlowMaxInt`.

Hence, we'll place this logic inside the constructor, like so:

```
public class SlowMaxInt implements MaxIntInterface {
  public SlowMaxInt() {
    this.numbers = new ArrayList<Integer>();
  }

  ...

}
```

But this couldn't work by itself. It has to be paired with a declaration that `numbers` is part of this class. Notice that this initializes the variable `numbers` that lives inside this particular instance. We create a new instance of `ArrayList<Integer>`, a collection type similar to Python list where we say the elements of this list will be integers (why Integer and not int?)

After this construction, inside any function of this class, we could refer to `numbers` either as `this.numbers` or as just `numbers` (when the name is not ambiguous).

Therefore, we are now ready to implement the remaining functions:

```
import java.util.*;

public class SlowMaxInt implements MaxIntInterface {
  // our variables
  private ArrayList<Integer> numbers;

  public SlowMaxInt() {
    this.numbers = new ArrayList<>();
  }

  public void add(int elt) {
    this.numbers.add(elt);
  }

  public int getMax() {
    Collections.sort(numbers);
    return numbers.get(numbers.size()-1);
  }
}
```

4.2 Improved MaxADT

In the previous implementation, our implementation requires $O(n)$ space to store a collection of n items. Moreover, although each `add` operation takes $O(1)$ time, a call to `getMax` takes time linear in the size of the collection—i.e., it takes $O(n)$ time if the collection has n items. *Can we do better?*

Our goal is to bring down both the space requirement and the time requirement. In fact, we aim to use $O(1)$ space and $O(1)$ time for all operations.

How can we possibly achieve that? A moment's thought reveals that we don't need to keep all the elements that have ever been added. Let's hash out a new plan:

Previous Version:

- Keep a list of data items (call this `numbers`).
- When the `add` operation is called, add the new item to `numbers`.
- When the `getMax` is called, scan `numbers` to find the largest item.

New Version:

- Keep only the largest data item seen so far (call this `maxEver`).
- When the `add` operation is called, compare with `maxEver` and replace it as necessary.
- When the `getMax` is called, just return `maxEver`.

Having laid out a plan, can we implement this together?

```
public class FastMaxInt implements MaxIntInterface {
  // our variables
  private int maxEver;

  public FastMaxInt() {
    this.maxEver = Integer.MIN_VALUE;
```

```
  }

  public void add(int elt) {
    if (elt > maxEver)
      maxEver = elt;
  }

  public int getMax() {
    return maxEver;
  }
}
```

5 A Bag Of Ints — A Bag Of Generic Items?

As another example, let us consider the following interface:

```
public interface IntBagInterface {
  // return current size
  public int size();

  // add a new item to the bag; a bag can have multiple copies
  // of the same item.
  public void add(int newItem);

  // remove one of the items (any of them) from the bag
  // and return the item removed
  public int remove();
}
```

There are many ways to implement this interface. For concreteness, the follow code shows an implementation that uses `ArrayList<Integer>`.

```
import java.util.*;

public class IntBag implements IntBagInterface{
  private List<Integer> entries; // contains all the entries in the collection

  public IntBag() {
    entries = new ArrayList<Integer>();
  }

  public int size() { return entries.size(); }

  public void add(int newItem) { entries.add(newItem); }

  public int remove() {  return entries.remove(0); } // remove the first elt
}
```

Note the use of `List<Integer>`. In this code, `List<Integer>` is an interface—it says this is an ADT of a list of integers. The specific implementation that we use is an ArrayList. This works because `ArrayList<Integer>` implements the interface we want: it implements `List<Integer>`.

5.1 What if we want a Bag of Doubles?

What if we want we wanted a `Bag` of doubles instead? Do we have to start over with `DoubleBagInterface` and `DoubleBag`? There must be a better way. One big hint is, we already know that we can tell `List` that we want a list of integers.

The magic sauce in this is called generics. We could declare an interface where we have a placeholder (a type parameter) for the type of our data. As an example:

```
public interface BagInterface<Item> {
  // return current size
  public int size();

  // add a new item to the bag; a bag can have multiple copies
  // of the same item.
  public void add(Item newItem);

  // remove one of the items (any of them) from the bag
  // and return the item removed
  public Item remove();
}
```

The most notable features are:

- `BagInterface<Item>` says this interface could be used as `BagInterface<Integer>`, `BagInterface<Double>`, `BagInterface<String>`, where the said `Item` is a type parameter.
- Later on, this generic type is used all over. In the function signature for `add`, we say that `newItem` has type `Item` (whatever it will be). Also, in the declaration for `remove`, the return type is `Item`.

One can update the IntBag implementation to support the new interface as follows:

```
import java.util.*;

public class Bag<Item> implements BagInterface<Item> {
    private List<Item> entries;

    public Bag() {
        entries = new ArrayList<>();
    }

    public int size() { return entries.size(); }

    public void add(Item newItem) { entries.add(newItem); }

    public Item remove() { return entries.remove(0); }
}
```

6 Finally, let's also support finding max

Did you know a class can implement multiple interfaces? This allows us to mix and match "capabilities." We will extend our running example of a BagADT to support finding the maximum element in the bag. A few steps are needed:

1. We hope to be able to write `class AwesomeBag<T> implements BagInterface<T>, MaxADT<T>`, so we'll need to make the MaxADT generic as well. This should be easy since we know how to do that already.

```
public interface MaxADT<T> {
  public void add(T elt);
  public T getMax();
}
```

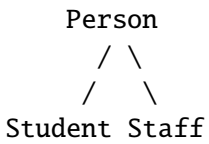
2. Now we'll update the add method to track the maximum. Again, we'll keep a private instance variable called `maxEver`. As written, the trouble is that we can't really compare the new element with `maxEver`. The generic type `T` is too generic: it doesn't say `T` must be comparable.
   To fix this, we'll instead write:

```
public class AwesomeBag<T extends Comparable<T>>
  implements BagInterface<T>, MaxADT<T> {
```

   This ensures that not every `T` can be used with this class—only types that are `Comparable` can be used as `T`.

6.1 Comparable what?

Avid Java programmers will have already spotted a problem with the above implementation. Let's be concrete here. Let's say we have the following classes:



Specifically, let's say we have `class Student extends Person implements Comparable<Person>`. So we expect `AwesomeBag<Person>` to be able to properly store instances of `Person` and `Student`. However, this is not the case. Why?

`Student` doesn't pass as `Comparable<Student>`. It only passes as `Comparable<Person>`.

We can fix this by revising how we declare `AwesomeBag`:

```
public class AwesomeBag<T extends Comparable<? super T>> ...
```

This means `T` can be whatever as long as `T` passes as `Comparable` of whatever that is a super class of `T`. That is, the thing inside `Comparable` derives to `T`.

That's it for today. Next week is inductive thinking and recursion.