Interfaces

Instructors: Sam McCauley and Dan Barowy

March 1, 2022

Admin

- Questions or comments?

Wrapping up memory

Autoboxing

- Sometimes we really want primitive types to act like objects
- For ex: `Vector<Integer>`
- Autoboxing: Java converts `int` to `Integer`, `char` to `Character`, etc., automatically.
- That's why this is OK. (The `Vector` really does store `Integer` objects)

```
Vector<Integer> vec = new Vector<Integer>;
vec.add(10);
```

Unboxing

- Java will also convert from objects (i.e. `Integer`) to primitive types (i.e. `int`) automatically:

```
Vector<Integer> vec = new Vector<Integer>;
vec.add(new Integer(10));
int x = vec.get(0);
```

Scope

Scope

- How long do *local variables* last in Java? That is: variables declared inside methods.

- (Not instance variables: they're accessible to the object the whole time)

Scope

- Any variable declared inside a method only lasts until *the end of that method*.
- Any variable declared inside a loop, (or if statement, etc.) only lasts until *the end of that loop*
- Actual rule: local variables only last inside the curly braces in which they were created

Interfaces

Interfaces: A Way to Standardize Behavior

- So far we've talked about creating classes
- Interfaces help us *group classes together*
- Allow us to create much more flexible code
- Object oriented programming is about much more than creating classes and objects! It's more about how classes interact.

Interface Example: Keeping Track of a Course

- Let's say we want to keep track of a course
- Course consists of objects of two types: `Student` and `TeachingAssistant`
- `Student` has instance variables:
  - `int age`, `String name`, `char grade`
- `TeachingAssistant` has instance variables:
  - `int age`, `String name`, `int numHours`
- Both have getters and setters. Let's take a look.

A Simple Task

- Let's say I want to go through all class participants (both students and TAs) and print out everyone who has age 20
- How can I do that?
  - Loop through students, check if age is 20, print if so
  - Same for TAs
- Let's try it

Redundancy

- These loops are exactly the same
- We're calling `getAge()` and `getName()` on each object. And each object is of a class type that does have these two methods. Why can't we do it in one loop?
- Need a way to put both types of object in *one array*.
    - Create an array of "things that have a `getName()` and `getAge()` method"

Interfaces

- A Java `Interface` is a *contract*
- An interface:
  - Defines methods (i.e. gives each methods' name, parameter, return types) that a class *must* implement
  - Kind of like a recipe for a class

Interfaces

- Allow us to group together classes: all classes that *implement* this `Interface` must have all of these methods
- Multiple classes can implement the same `Interface`
- We interact with objects using methods. So if multiple objects have the same interface, we can interact with them in a unified way.

Using Interfaces for Our Example

- Students and TAs both are people—so they both have `getName()` and `getAge()` methods
- Let's write a `Person` interface: a contract for these methods. Every class type implementing `Person` must have a `getName()` and `getAge()` method
- Then let's tell Java that `Student` and `TeachingAssistant` both *implement* `Person`

Combining Into One Loop

- Let’s refactor our array what type of objects does it store?
    - The items in our array have a `getName()` and `getAge()` method
    - So…it stores items of type `Person`
    - Let’s try it

Interfaces Syntax

- A class can *implement* an interface by providing code for each required method.
- If we have code that only depends on these methods, the code should work for objects of any class that implements that interface
- If the methods aren't all implemented, Java gives an error.

Interface Types

- It's OK to create a variable of interface type (i.e. of type `Person`)
- We just created an array where the type was of `Person` for example
- But we cannot *instantiate* any object as instance type
  - Cannot say: `Person p = new Person();` for example
- Why not?
  - `Person` does not have a constructor! (It doesn't even have specific instance variables; Java has no idea what to create)
- Short version: when *instantiating* need a specific class type. But can store it as any interface that the class type implements.

Lists and Abstract Data Types

Interacting with Data

- In Java, we store our data in objects
- We interact with that data using methods
- Idea of an abstract data type: the *methods* are what's important, not the details of how the data is stored
- Example: let's say we have a list of data. We want to call methods like `contains` on it. Does it matter what the method is for expanding the underlying array? (Or if there's an array at all?)

List: first ADT Example

- `Vector` is a super useful class
- You've all probably used something similar in the past.
    - Did what you used have an array as a back end? Who knows. (Who cares?)
- Let's define exactly what we want a list to do

`List`

A `List` should handle operations:

- `get(i)`
- `add(E)`, `set(i, E)`
- `contains(E)`, `indexOf(E)`
- `size()`
- Etc.

Idea: let's make a `List` interface. Any time you interact with data using only these operations (e.g. on the wordgen lab), can just store a `List`

- We'll come back to this tomorrow

Time and Recursion Review

What is the running time of `indexOf` for a `Vector`?

- Let's look at the indexOf operation
- What is its big-O running time in the *worst case* in terms of the *size of the vector*?
  - Let n be the size of the vector
  - We'll write our answer as $O(g(n))$:
  - In this case, we can use $g(n) = n$. So the worst case is $O(g(n))$.
- What is its big-O running time in the *best case*?
  - Best case means *best data*. Not best n!
  - In the best case, we find the element immediately. $O(1)$.

Another running time example

- Let's assume we run `indexOf`, and we *don't* find the item
- What is the best case running time?
- Answer: it doesn't matter what the *contents* of the `Vector` are; we loop through the entire `Vector` every time
- Best case running time is $O(n)$.
- Reminder: this is an *upper bound*. In the best case, we take at most $O(n)$ time.