Lecture 15 — Interfaces

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 8 of the textbook
- Creating Interfaces
- Implementing interfaces
- Extending interfaces

Multiple Inheritance

- Java does not allow multiple inheritance
- Suppose we have classes A and B
- A class C cannot extend both A and B
- Why is this?

Multiple Inheritance

- It is to avoid ambiguity
- If two classes have the same method and we inherit from both, which version of the method do we get?
- If two classes have the same field with different types/access, which one do we get?
- Java simply avoids this issue by not allowing multiple inheritance

BasicSafeArray

```
class A {
    int x;
    void doSomething() {
        // something
    }
}

class B {
    float x;
    void doSomething() {
        // something else
    }
}

class C extends A, B {
    // ??
}
```

Abstract Multiple Inheritance

```
abstract class A {
    abstract void doSomething();
}

abstract class B {
    abstract void doSomething();
}

class C extends A, B {
    // override doSomething()
}
```

- Sometimes, multiple inheritance might make sense
- Recall abstract classes
- An abstract method is a method that is not implemented and must be overridden
- If the only collisions from multiple inheritance are identical abstract methods, this should not cause an issue

Interfaces

```
class D extends A implements B { ... }

class E implements B { ... }

class F implements B, C { ... }
```

- This is where *interfaces* come in
- An interface is like a class that only contains abstract methods
- The issues of multiple inheritance do not exist for interfaces, so Java allows you to "inherit" from many of them
- When we "inherit" from an interface, the keyword is `implements`
- A class can extend up to one superclass and implement any number of interfaces

Interfaces

```
public interface Example {
    void exampleMethod1(int param);
    double exampleMethod2(String s, int x);
}
```

- Interfaces are defined similarly to classes except we use the keyword `interface`
- There should be one top-level interface with the same name as the file
- Interface members are public by default
- Note that non-public members don't fit the philosophy of an interface: an interface is an abstract collection of methods without any implementation specifics
- An interface is like a contract that a class can fulfill

HasLegs Example

```
public interface HasLegs {
    int countLegs();
}
```

- This interface represents things that have a number of legs
- This sort of concept is usual for an interface
- An interface usually represents a notion or trait
- A class, by comparison, usually represents a type of thing
- A class may *implement* some interfaces
- Let's see an example usage of this interface

HasLegs Example

```java
class Chair implements HasLegs {
    public int countLegs() {
        return 4;
    }
}

class Person implements HasLegs {
    public int countLegs() {
        return 2;
    }
}

class Spider implements HasLegs {
    public int countLegs() {
        return 8;
    }
}
```

HasLegs Example

```java
public class CountLegs {
    public static void main(String[] args) {
        HasLegs[] things = new HasLegs[3];
        things[0] = new Chair();
        things[1] = new Person();
        things[2] = new Spider();
        int sum = 0;
        for (int i = 0; i < things.length; i++) {
            sum += things[i].countLegs();
        }
        System.out.println("Total number of legs: " + sum);
    }
}
```

- Notice that the interface `HasLegs` can be used as a valid variable/array type, similarly to a class! Polymorphism works too

MakesSounds Example

```
public interface MakesSounds {
    String sound();
}
```

- Let's introduce another interface
- It is possible to implement multiple interfaces
- We will see an example

MakesSounds Example

```
abstract class Insect implements HasLegs {
    @Override
    public int countLegs() {
        return 6;
    }
}

class Cricket extends Insect implements MakesSounds {
    @Override
    public String sound() {
        return "Chirp";
    }
}

class SqueakyChair implements HasLegs, MakesSounds {
    @Override
    public int countLegs() {
        return 4;
    }

    @Override
    public String sound() {
        return "Squeak";
    }
}
```

MakesSounds Example

```
public class SoundExample {
    public static void main(String[] args) {
        MakesSounds[] things = {new Cricket(), new SqueakyChair()};
        for (MakesSounds thing : things)
            System.out.println(thing.sound());
        HasLegs[] legs = {new Cricket(), new SqueakyChair()};
        for (HasLegs leg : legs)
            System.out.println(leg.countLegs());
    }
}
```

- `Cricket` and `SqueakyChair` implement two interfaces
- Inheritance via `extends` can be used alongside interfaces
- Because of multiple inheritance via interfaces, it might be nice to be able to do something like `HasLegsAndMakesSounds[] things = ...`

HasLegsAndMakesSounds Example

```
public interface HasLegsAndMakesSounds extends HasLegs, MakesSounds {
    // This interface has no methods of its own.
    // It inherits the methods from its two parent interfaces.
}
```

- Interfaces can extend other interfaces
- Interfaces can extend more than one other interface
- In this case, we don't add any extra methods, but we could if needed

HasLegsAndMakesSounds Example

```
abstract class Insect implements HasLegs {
    @Override
    public int countLegs() {
        return 6;
    }
}

class Cricket extends Insect implements HasLegsAndMakesSounds {
    @Override
    public String sound() {
        return "Chirp";
    }
}

class SqueakyChair implements HasLegsAndMakesSounds {
    @Override
    public int countLegs() {
        return 4;
    }

    @Override
    public String sound() {
        return "Squeak";
    }
}
```

HasLegsAndMakesSounds Example

```
public class LegsAndSoundsExample {
    public static void main(String[] args) {
        HasLegsAndMakesSounds[] things = {new Cricket(), new SqueakyChair()};
        for (HasLegsAndMakesSounds thing : things) {
            System.out.println(thing.sound());
            System.out.println(thing.countLegs());
        }
    }
}
```

- Notice that `Cricket` could implement `HasLegsAndMakesSounds` even though `Insect` already implemented `HasLegs`
- An interface is just a bundle of methods you are required to override, so this is generally fine

Duplicated Variables

```
interface A {
    int x = 1;
}

interface B {
    double x = 2.0;
}

interface C extends A, B {
    // What is the type of C.x?
}

public class DuplicateFields {
    public static void main(String[] args) {
        System.out.println(C.x);
    }
}
```

- Interfaces can have (`static final`) fields
- In the case of multiple interface inheritance, any duplicated variable names will cause a compilation error
- The use of these is rare as a result

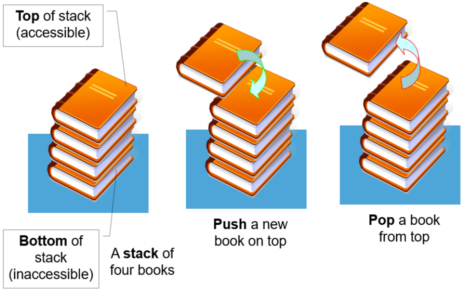
Mid-lecture Break

Default Methods

```java
interface HasId {
    default int getId() {
        return 0;
    }
}

public class Admin implements HasId {
    // No need to override getId()
}
```

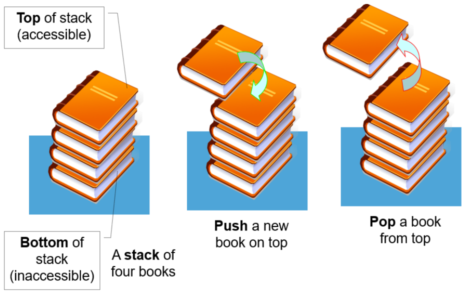
- In modern versions of Java (since JDK 8), it is possible for interfaces to define an implementation for methods
- Such a method is called a `default` method
- They are restricted in what they can do since interfaces have no instance fields to modify
- They can also lead to multiple inheritance errors and are thus typically for use in specific situations only
- They are beyond the scope of this course, but are mentioned here for completeness

Stack Interface


Top of stack (accessible)
Bottom of stack (inaccessible)
A **stack** of four books
Push a new book on top
Pop a book from top

- The previous lecture on exceptions used an example implementation of a stack
- Recall that the stack allows us to push and pop elements from/to the top
- Think of a stack of papers, or a stack of pancakes if you're hungry
- Our stack will store a stack of strings

Stack Interface



- Previously, we implemented the stack using an array
- There are many ways to implement a stack of strings
- Fixed size array; array grows as needed; ArrayList; LinkedList; many more
- However, the essence of a stack stays the same; any code that uses a stack doesn't need to know the details
- This is where an interface can help as write expressive code

Stack Interface

```
public interface StringStack {
    String pop();
    void push(String s);
    boolean isEmpty();
}
```

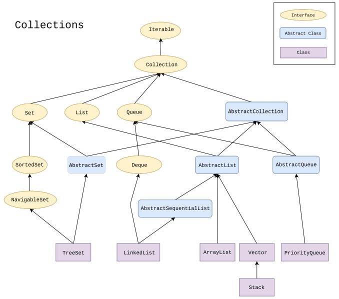
- This set of three methods gives us a basic contract
- Any stack of strings should fulfill this
- Note that we did not use `isEmpty()` in the previous lecture

Stack Interface

```
public class ArrayStringStack implements StringStack {
    private String[] stack;

    public ArrayStringStack() {
        stack = new String[0];
    }

    @Override
    public String pop() {
        // removes this.stack[0] and returns it
    }

    @Override
    public void push(String s) {
        // Resizes this.stack with s at position 0
    }

    @Override
    public boolean isEmpty() {
        return stack.length == 0;
    }
}
```

- This is one example implementation (different to the previous lecture's)
- Let's open up the lecture code folder and have a look at the details

Java Collections



Taken from `https://en.wikipedia.org/wiki/Java_collections_framework#/media/File:Java.util.Collection_hierarchy.svg`

- The Java API makes heavy use of interfaces and abstract classes
- The `Collection` hierarchy in particular is notable
- A `Collection` is an interface representing any group of things

Java Collections

- It has subinterfaces for lists, queues, and more
- `ArrayList` implements the list interface
- Amusingly, there is no stack interface, but there is a deque interface
- A stack class (not interface) existed before they introduced interfaces, and now it cannot be removed due to backwards compatibility
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/package-summary.html`