

The “Implements” Relation and Default Implementations

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Wednesday, Feb. 8, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

General Announcements

1 Exam #1 - Feb. 22, 2023

- ▶ Please complete the Lockdown Browser Practice Quiz as part of your participation grade when it opens up. More information will be announced on Canvas.
- ▶ Will post review slides soon!

Homework Assignment

- 1 Watch the various videos posted on your lab Canvas page.
- 2 Programming assignment 1 is posted on your lab Canvas page.

The “Implements” Relation

- ▶ The `implements` relation may hold between a `class` and an `interface`
- ▶ If `C implements I` then class `C` contains code for the behavior specified in interface `I`
 - ▶ This means `C` has method bodies for instance methods whose contracts are specified in `I`
 - ▶ The code for `C` looks like this:

```
public class C implements I {  
  
    // bodies for methods specified in I  
    ...  
  
}
```

The “Implements” Relation

- ▶ The `implements` relation allows you to separate contracts from their implementations – a **best practice** for component design.
- ▶ The Java compiler checks that `C` contains bodies for the methods in `I`, but does not check that those bodies correctly implement the method contracts!

Best Practice

- ▶ Separating our contracts in our `interface` from the code in our implementation is a best practice
 - ▶ It helps us hide information from the client
- ▶ Old versions of Java (prior to Java 8) did not allow code in the `interface`
- ▶ New versions of Java does allow it as a **default implementation**
 - ▶ This needs to be done carefully, and only in certain circumstances

Default Implementations

- ▶ Java now allows code in our `interface` as long as it is a `default` or `static` methods
 - ▶ **NOTE:** While `static` methods can be added to Java `interfaces`, we will not be adding them to the ones we create.
- ▶ `default` methods are allowed
 - ▶ Still no `private` data fields
 - ▶ We can have local variables declared inside the method
 - ▶ How can we add code without knowing it's `private` data?
- ▶ We only use this to add secondary methods to our `interface`

Primary vs. Secondary Operations

- ▶ A **primary** operation is a method that needs access to the **private** data of a class
 - ▶ Without them, we wouldn't be able to do anything useful because we couldn't access the state of the object
 - ▶ **Examples:** **push**, **pop**, and **depth** are primary operations for a **Stack**
 - ▶ Recall, **Stacks** are LIFO
- ▶ A **secondary** operation is a method that does not need direct access to the **private** data of a class.
 - ▶ It can get access through other methods
 - ▶ If a secondary method didn't exist, we could still perform that operation using the primary methods
 - ▶ **Example:** A **top** operation that returns the top item of the **Stack** without removing it from the **Stack**

Secondary Operations

- ▶ Secondary operations don't **need** to access the **private** data directly
 - ▶ How do we know?
 - ▶ Can we use the primary methods available to complete the task if this secondary operation did not exist?
- ▶ Consider our **top** example
 - ▶ If **top** was not a method, could we implement it using the primary methods?
 - ▶ Yes! How?
 - ▶ We don't need it, but it would be helpful to just call `<stack_var>.top()` instead.
- ▶ Secondary operations can be implemented using the primary operations.

Default Implementations

- ▶ Java now allows code in our **interface** as long as it is a **default** or **static** methods
 - ▶ **NOTE:** While **static** methods can be added to Java **interfaces**, we will not be adding them to the ones we create.
- ▶ **default** methods are allowed
 - ▶ Still no **private** data fields
 - ▶ We can have local variables declared inside the method
 - ▶ How can we add code without knowing it's **private** data?
 - ▶ We use the methods provided by the **interface**
- ▶ We only use this to add secondary methods to our **interface**
 - ▶ Our code that we add only uses the primary methods
 - ▶ Never needs to access the **private** data directly

Consider...

- ▶ You are working on a large project, with an existing `interface`
- ▶ The `interface` would be easier to use if there was an additional method available
- ▶ You add the method to the `interface`
- ▶ The project is suddenly full of errors!
 - ▶ Why?
- ▶ How can we add a method to an `interface` without breaking existing implementations?

Default Implementations

- ▶ To add your new method, add the method with the keyword `default`.
- ▶ Then provide the implementation using the primary methods to access the `private` data
- ▶ When an implementation does not override this new method, the compiler will use the `default` implementation provided
- ▶ Since you used primary methods to implement your `default` implementation, your code will compile
 - ▶ The existing implementations already had those methods defined, so nothing needs to change

Default Implementations

- ▶ It is easy to add a method with a `default` implementation to your `interface`
- ▶ It is the same as your primary methods, with two key differences
 - 1 Use the keyword `default` at the beginning of the signature
 - 2 Add a body to the method that uses primary operations to complete the secondary operation

```
public interface myInterface{  
  
    public int primaryOp1();  
    public int primaryOp2();  
  
    default public int secondaryOp() {  
        return primaryOp1() + primaryOp2();  
    }  
  
}
```

IntegerStack Interface

Primary Operations

```
public interface IntegerStack {  
  
    int MAX_DEPTH = 100;  
  
    void push(Integer x);  
    Integer pop();  
    void clear();  
    int depth();  
  
}
```

Recall: How to Specify this Interface?

```
/**  
  Stack is abstractly a string of integers.  
  
  Initialization ensures:  
    Stack contains nothing, i.e., an empty string <>.  
  
  Constraints: length of a self <= MAX_DEPTH  
*/  
public interface IntegerStack {  
    public static final int MAX_DEPTH = ...;  
    ...  
}
```

Recall: How to Specify this Interface?

```
public interface IntegerStack {  
    ...  
  
    /**  
        @pre    length of self, |self| < MAX_DEPTH  
        @post   self = [x added to the front of #self]  
    */  
    void push(Integer x);  
  
    /**  
        @pre    length of self, |self| > 0  
        @post   self = [pop() removed from the front of #self]  
    */  
    Integer pop();  
  
    ...  
}
```

Implementation #1

```
/**
 * @invariant -1 <= top < MAX_DEPTH
 * @correspondence self = [ reversed string of Integers from contents[0] to contents[top] ]
 *
 * Note: if top is set initially to be < 0
 *       then this would correspond to empty string
 */
public class Stack1 implements IntegerStack {

    private Integer[] contents;
    private int top;

    /* code for primary methods only! */
    public Stack1() {
        contents = new Integer[MAX_DEPTH];
        top = -1;
    }

    public void push(Integer x) {
        contents[top++] = x;
    }

    public Integer pop() {
        Integer x = contents[top];
        top--;
        return x;
    }

    public int depth() {
        return top+1;
    }
}
```

Implementation #2

```
/**
 * @invariant 0 <= top <= MAX_DEPTH
 * @correspondence self = [ string of Integers from contents[top] to contents[MAX_DEPTH - 1] ]
 *
 * Note: if top is set initially to be > MAX_DEPTH - 1
 *       then this would correspond to empty string
 */
public class Stack2 implements IntegerStack {

    private Integer[] contents;
    private int top;

    /* code for primary methods only! */
    public Stack2() {
        contents = new Integer[MAX_DEPTH];
        top = MAX_DEPTH;
    }

    public void push(Integer x) {
        contents[top--] = x;
    }

    public Integer pop() {
        Integer x = contents[top];
        top++;
        return x;
    }

    public int depth() {
        return MAX_DEPTH - top;
    }
}
```

IntegerStack Interface

Secondary Operation: `top`

```
public interface IntegerStack {  
  
    int MAX_DEPTH = 100;  
  
    void push(Integer x);  
    Integer pop();  
    void clear();  
    int depth();  
  
    default public Integer top() {  
        Integer x = pop();  
        push(x);  
  
        return x;  
    }  
}
```

IntegerStack Interface

- ▶ Nothing needs to change in our implementations
 - ▶ Or any others that were added without our knowledge
- ▶ Every stack provided the code for `push` and `pop`, so the `top` operation works already.
- ▶ If a specific implementation has a more efficient way of implementing `top`, it can do so and override the `top` operation that is provided in the `interface`

Factoring Out Common Code

- ▶ Method bodies that can be written once – and work for any implementation of `IntegerStack` because they are programmed to that interface – have been factored out into a default implementation
- ▶ This leaves only constructors and primary operations to be implemented in `Stack1`, `Stack2`, and future classes that implement `IntegerStack`

Issues

- ▶ What if we want to add an operation to an `interface` we can't edit?
 - ▶ Java libraries
- ▶ What if we want to add functionality to an `interface`, but don't want every implementation to have that functionality.
 - ▶ **Ex:** We have a `Customer` interface and implementation, now we want a `DeliveryCustomer` interface and implementation with extra functionality
- ▶ We can use inheritance to extend the `interface`
- ▶ Inheritance works for interfaces, just like it works for classes
 - ▶ Your new implementations will need to implement the extended `interface`, not the original `interface`
 - ▶ Implementations of the extended `interface` have to provide code for operations in the base interface and the extended `interface`

General Announcements

1 Exam #1 - Feb. 22, 2023

- ▶ Please complete the Lockdown Browser Practice Quiz as part of your participation grade when it opens up. More information will be announced on Canvas.
- ▶ Will post review slides soon!

Homework Assignment

- 1 Watch the various videos posted on your lab Canvas page.
- 2 Programming assignment 1 is posted on your lab Canvas page.

Summary

- 1 The “Implements” Relation
- 2 Default Implementations