

The “Extends” Relation and Inheritance

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Monday, Feb. 13, 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.

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**

The “Extends” Relation

- ▶ The **extends** relation may hold between:
 - ▶ Two interfaces or
 - ▶ Two classes
- ▶ In either case, if **B extends A** then **B inherits** all the methods of **A**
 - ▶ This means **B** implicitly starts out with all the method contracts (for an **interface**) or all the method bodies (for a **class**) that **A** has
 - ▶ **B** can then *add more* method contracts (for an **interface**) or method bodies (for a **class**)
 - ▶ **B** can **override** the methods.

Caveats About Java Classes

- ▶ “If **B** **extends** **A** then **B** **inherits** all the methods of **A**”
 - ▶ Constructors **are not** inherited
 - ▶ So in the situation above, the class **B** must include bodies for any constructors that are expected, even if they would be identical to those of **A**
 - ▶ The bodies of the constructors in **B** generally would simply invoke the constructors of **A**, which is done using the special notation **super(...)**
 - ▶ Static methods **are** inherited

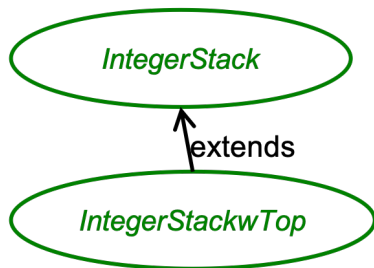
Interface Inheritance Example

IntegerStackwTop Interface

```
public interface IntegerStackwTop extends IntegerStack {  
  
    /**  
     * @pre length of self, |self| > 0  
     * @post top = [ the value on top of the stack ] and  
     *         self = #self  
     */  
    Integer top();  
  
}
```

Interface Inheritance Example

`IntegerStackwTop` `Interface`



`push`
`pop`
`depth`
`getMaxDepth`
`clear`

+

`top`

=

`push`
`pop`
`depth`

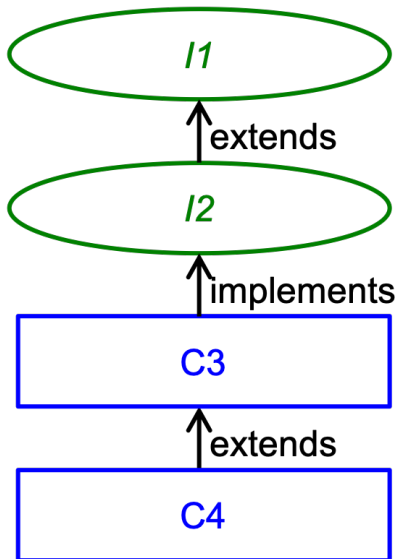
`getMaxDepth`
`Clear`
`top`

Note: `IntegerStackwTop` actually has all the methods in `IntegerStack`, even though their contracts are in two separate `interfaces`.

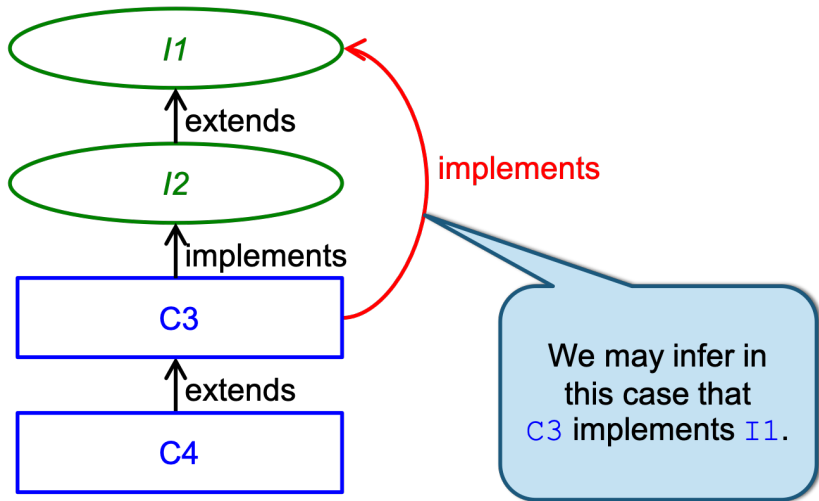
Extended Interfaces

- ▶ Now we can add new methods to an **interface** we can't edit
- ▶ When we implement our extended **interface**, we need to provide code for the new operations
 - ▶ What if the new operation is a secondary operation that uses the primary operation?
 - ▶ The code will be the same for any implementation
 - ▶ We can still create **default** methods in the extended **interface**
- ▶ We will need to create an implementation of the extended **interface**
 - ▶ Provides code for the primary methods
 - ▶ The **decorator pattern** helps here (a later lecture)

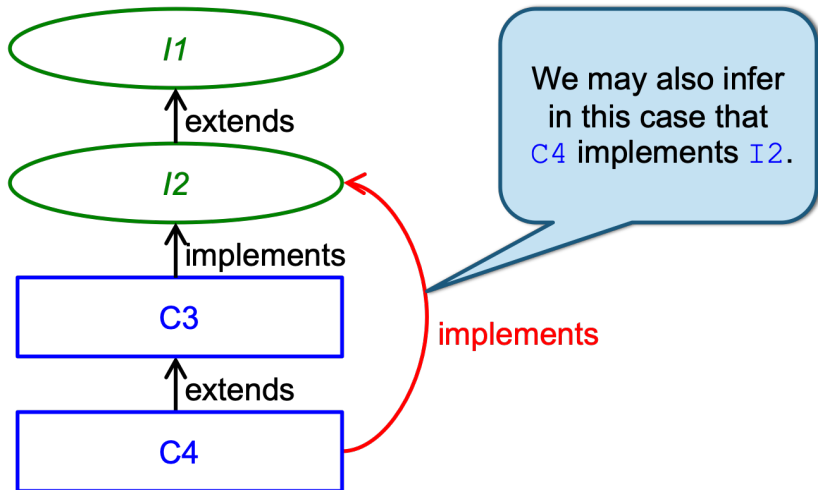
“Implements” May Be Inferred



“Implements” May Be Inferred



“Implements” May Be Inferred



The Ubiquitous Class: `Object`

- ▶ Every **class** in Java extends `Object`, which is a special built-in class that provides default implementations for the following instance methods (among a few others that are not so important):

```
boolean equals(Object obj)
int hashCode()
String toString()
```

- ▶ We can **override** them to provide more meaningful functionality.

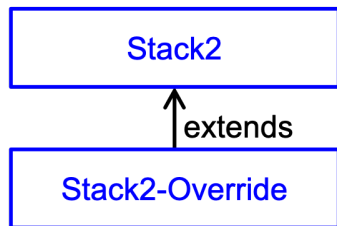
```
@Override
public String toString() {
    ...
}
```

- ▶ This works because of our inheritance structure and our extends relationship

Overriding Object Methods

- ▶ While the methods provided by `Object` can be written only using primary operations. . .
 - ▶ You can't provide a `default` implementation in the `interface`
 - ▶ The `interface` does not `extend Object`, because an `interface` cannot extend a `class`

Which Method Body is Used?



...
push

...

?

push

This raises the question:
Which method body for `push` is
used when `push` is called in a client
program?

Polymorphism

- ▶ Java and other object-oriented languages decide which method body to use for any call to an instance method based on the **dynamic type** of the **receiver**
 - ▶ This type, because it is the *class* of the constructor, is always a *class* type
- ▶ This behavior for calling methods is known as **polymorphism**: “having many forms”

Abstract Classes

- ▶ Java permits you to write a kind of “partial” or “incomplete” class that contains bodies for some but (typically) not all of the methods of the interfaces it claims to implement
- ▶ Such a class is called an **abstract class**:

```
public abstract class AC implements I {  
    ...  
}
```

- ▶ Because some methods still might not have bodies, Java will not let you **instantiate** an **abstract** class; that is, you cannot use an **abstract** class like a normal class and create a new object from it.

Abstract Classes

- ▶ Abstract classes can include data fields, and code for methods
 - ▶ Even non-default methods
- ▶ You can then extend from the abstract class to inherit it's data and methods like any other class
- ▶ These were more helpful when Java did not have default implementations
 - ▶ We could add secondary methods into the abstract class to factor out common code
- ▶ Now they are less common but can still be helpful

When to Use Abstract Classes

- ▶ If you wanted to define data fields that every subclass must contain
- ▶ Provide methods that use those data fields that every subclass can inherit
- ▶ Many design patterns will refer to using abstract classes
 - ▶ Partially because abstract classes exist in many languages that don't have **interfaces**
 - ▶ An abstract class with no data fields can be used in a similar manner as an **interface**
 - ▶ Declare primary methods as abstract
 - ▶ Add secondary methods that call primary methods
 - ▶ Subclasses override the primary methods and inherit the secondary methods
 - ▶ **Note:** We will revisit this when we talk about design patterns later in the semester!

Abstract Class vs. Interface

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance .	Interface supports multiple inheritance .
3) Abstract class can have final, non-final, static and non-static variables .	Interface has only static and final variables .
4) Abstract class can provide the implementation of interface .	Interface can't provide the implementation of abstract class .
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
9) Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

Src: <https://www.javatpoint.com/>

difference-between-abstract-class-and-interface

Important Reasons¹

Important Reasons For Using Interfaces

- ▶ Interfaces are used to achieve abstraction
- ▶ Designed to support dynamic method resolution at run time
- ▶ It helps you to achieve loose coupling
- ▶ Allows you to separate the definition of a method from the inheritance hierarchy

Important Reasons For Using Abstract Class

- ▶ Abstract classes offer default functionality for the subclasses.
- ▶ Provides a template for future specific classes
- ▶ Helps you to define a common interface for its subclasses
- ▶ Abstract class allows code reusability

¹<https://www.guru99.com/interface-vs-abstract-class-java.html>

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 “Extends” Relation
- 2 Inheritance
- 3 Abstract Classes