

Information Hiding and Interfaces

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Wednesday, Feb. 1, 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.

Last Class

- 1 Classes and Objects
- 2 Invariants
- 3 Good Practices
- 4 Types of Objects
- 5 Practice with Contracts

Homework Assignment

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

Three Questions to Answer Today

- 1 What is an interface?
- 2 Why interfaces?
- 3 How to develop and use interfaces in Java?

What is an interface?

- ▶ An interface is where interactions occur
 - ▶ **Ex 1:** *User interface* is the interface between the user and the program
 - ▶ **Ex 2:** *Light switch* is the interface between a person and the electrical system
- ▶ An interface
 - ▶ Describes *what* classes should do
 - ▶ Does not describe *how* they should do it
- ▶ An interface expresses some coherent concept (e.g., stacks, queues, sets)
 - ▶ Provides the methods available
 - ▶ Contracts for those methods
- ▶ Multiple classes (solutions) may implement the same interface
- ▶ Clients can ignore the implementation and rely on just interfaces

Consider...

- ▶ We don't really have interfaces in C++
- ▶ But we had our header files
 - ▶ `MyClass.h` - gave a description of the class
 - ▶ `MyClass.cpp` - gave an implementation of the class
- ▶ Interfaces in Java:
 - ▶ Provides the description and contracts
 - ▶ Separate code file give the implementation
- ▶ Major difference:
 - ▶ In Java, we can have multiple classes that implement the interface
 - ▶ They don't have to have the same name

Recall “List” Data Structure

- ▶ `List<String> stringList = new ArrayList<>();`
- ▶ `List` vs `ArrayList`?

Recall “List” Data Structure

- ▶ `List<String> stringList = new ArrayList<>();`
- ▶ `List` vs `ArrayList`?
- ▶ `List` is an interface in Java
 - ▶ It gives the contracts and methods that exists for any `List`
 - ▶ `add`, `remove`, `get`, `set`, `size`, etc

Recall “List” Data Structure

- ▶ `List<String> stringList = new ArrayList<>();`
- ▶ `List` vs `ArrayList`?
- ▶ `List` is an interface in Java
 - ▶ It gives the contracts and methods that exists for any `List`
 - ▶ `add`, `remove`, `get`, `set`, `size`, etc
- ▶ `ArrayList` is a class that implements the `List` interface
 - ▶ It provides the code for the methods

Recall “List” Data Structure

- ▶ `List<String> stringList = new ArrayList<>();`
- ▶ `List` vs `ArrayList`?
- ▶ `List` is an interface in Java
 - ▶ It gives the contracts and methods that exists for any `List`
 - ▶ `add`, `remove`, `get`, `set`, `size`, etc
- ▶ `ArrayList` is a class that implements the `List` interface
 - ▶ It provides the code for the methods
- ▶ Other implementations exist such as `LinkedList`
 - ▶ Both `ArrayList` and `LinkedList` have the methods specified in the `List` interface

Why use interfaces?

- ▶ Information Hiding
- ▶ Separation of Concerns

From Classes to Interfaces

- ▶ Aren't classes (and objects) enough to solve the information hiding problem?

From Classes to Interfaces

- ▶ Aren't classes (and objects) enough to solve the information hiding problem?
 - ▶ No. They don't hide well.
 - ▶ Users are still burdened because they have to look at all the class details.
 - ▶ When a class representation and methods change, user understanding is affected

From Classes to Interfaces

- ▶ Aren't classes (and objects) enough to solve the information hiding problem?
 - ▶ No. They don't hide well.
 - ▶ Users are still burdened because they have to look at all the class details.
 - ▶ When a class representation and methods change, user understanding is affected
 - ▶ JavaDocs and contracts provide only a partial solution
 - ▶ precise explanations of `public` class methods may refer to `private` data and code (not desirable)
 - ▶ We had issues with writing our contracts
 - ▶ Do we, or do we not refer to the names of variables?

From Classes to Interfaces

- ▶ There is another fundamental issue
- ▶ A class is a particular solution to a problem
 - ▶ It has a specific private data representation
 - ▶ It has methods based on that private data representation
- ▶ In general, there are many solutions with different trade-offs for the same problem (e.g., sorting and searching)
 - ▶ This is what you did (or will be doing) in CPSC 2120!

Interfaces and Multiple Classes

- ▶ Why are there so many sorting algorithms?

Interfaces and Multiple Classes

- ▶ Why are there so many sorting algorithms?
 - ▶ Performance trade-offs.
 - ▶ Speed vs memory efficiency
- ▶ Why would we want multiple classes that implement the same interface?

Interfaces and Multiple Classes

- ▶ Why are there so many sorting algorithms?
 - ▶ Performance trade-offs.
 - ▶ Speed vs memory efficiency
- ▶ Why would we want multiple classes that implement the same interface?
 - ▶ Alternative class implementations provide clients with choices.
 - ▶ Clients can pick an interface and then pick a class implementation that best suits their needs.

Performance Trade-Offs

- ▶ In general, there is no one “best” solution to a problem.
 - ▶ Some more space conscious and some more time conscious.
 - ▶ A subset of methods faster in one; a different subset faster in another.
 - ▶ Some better on average, but some better in worst case.
 - ▶ Some more predictable; some more efficient.

Performance Trade-Offs

- ▶ We have an interface: `Grid`
 - ▶ Stores the characters in a grid
 - ▶ Provides methods such as `addMarker`, `whatsAtPos`, and other useful methods
- ▶ We have 2 implementations of the interface:
 - 1 `GridFast` uses a 2D array of characters
 - ▶ Very fast to check a particular position
 - ▶ Very memory intensive
 - 2 `GridMem` uses a `List` of `<character, row, column>`
 - ▶ Takes more time to check a particular position
 - ▶ Much more memory efficient (until grid fills up)
- ▶ How the grid is implemented is different, but since they implement the same interface, they have the same `addMarker` and `whatsAtPos` methods!

Performance Trade-Offs

- ▶ We have a program that uses `Grid`, but we won't know until runtime if we are more concerned about speed or memory

```
Grid aGrid;  
if (...) { // Memory usage is key  
    aGrid = new GridMem();  
}  
else { // Execution performance is key  
    aGrid = new GridFast();  
}  
  
// Nothing else has to change in our program  
aGrid.addMarker(...);
```

Performance Trade-Offs

- ▶ Remember, our interface `Grid` defines the methods that must be available
 - ▶ This means `GridMem` and `GridFast` has the same list of methods available to them.
- ▶ Later when we call `aGrid.methodA()` we know it will work because `methodA` is defined for both `GridMem` and `GridFast`
- ▶ Our client code only cares about the `Grid` interface, so you can swap the implementation at any time

How to use interfaces?

- ▶ We know what an interface is.
- ▶ We know why we use interfaces.
- ▶ How do we create and use interfaces?

Note on Examples That Follow

- ▶ Intended to illustrate the mechanics of declaring and using interfaces in Java.
- ▶ They don't motivate the principle of information hiding from the preceding slides!
- ▶ You will see more illustrative examples later.

Java Syntax for Interfaces

- ▶ An interface
 - ▶ Describes *what* classes should do
 - ▶ Does not describe *how* they should do it

- ▶ **Example:**

```
public interface Salaried {  
    void setSalary(double d);  
    double getSalary();  
}
```

- ▶ To satisfy this interface, a class must provide `setSalary` and `getSalary` methods with
 - ▶ matching signatures (checked by compiler)
 - ▶ matching behaviors (up to you)

Declaring an Interface

- ▶ Looks like a class definition, except:
 - ▶ Keyword `interface` replaces `class`
 - ▶ Methods have no body
 - ▶ No constructors
- ▶ Like a `class`, an `interface` can contain
 - ▶ **Fields:**
 - ▶ Must be `public static final` (ie constants)
 - ▶ Do not include our `private` fields
 - ▶ **Methods:**¹
 - ▶ Must be `public abstract` (ie bodiless) or `default` (future lecture)
 - ▶ Can not be `final`
 - ▶ The `interface` itself is `public` or *package* visible

¹While you can have `static` methods in an `interface`, we won't use it in this class

Example #1

```
public interface Salaried {  
    void setSalary(double d);  
    double getSalary();  
}
```

Example #2

```
interface Voter {  
    int MINIMUM_AGE = 18;  
    Voter(short age); //compile-time error  
    void Register(District d);  
    boolean isRegistered();  
}
```

Implementing an Interface

- ▶ Declare a class that *implements* the interface:

```
class Employee implements Salaried { ... }
```

- ▶ Supply definitions for *all* interface methods

```
public void setSalary(double d) {  
    ...  
}
```

```
public double getSalary() {  
    ...  
}
```

- ▶ **Note:** `public` modifier of method can *not* be omitted in `class` definition (even though it is omitted in `interface`)
- ▶ `class` can declare more methods than required by `interface`
- ▶ `class` can implement more than one `interface`

Good Practice: Naming Interfaces

- ▶ How should interfaces be distinguished from classes in their names?
- ▶ One approach:
 - ▶ Classes end in “_Impl” (or “_Realiz”, ...)
 - ▶ E.g., `Stack_Interface.java` vs. `Fast_Stack_Impl.java`
- ▶ Microsoft approach:
 - ▶ Interfaces start with “I”
 - ▶ E.g., `IStack.java` vs. `Stack.java`
- ▶ Java approach:
 - ▶ No difference, both are nouns or adjectives

Instantiating an Interface

- ▶ The **declared type** of a variable can be an **interface**

```
Salaried payee; //ok
```

- ▶ But interfaces cannot be instantiated directly

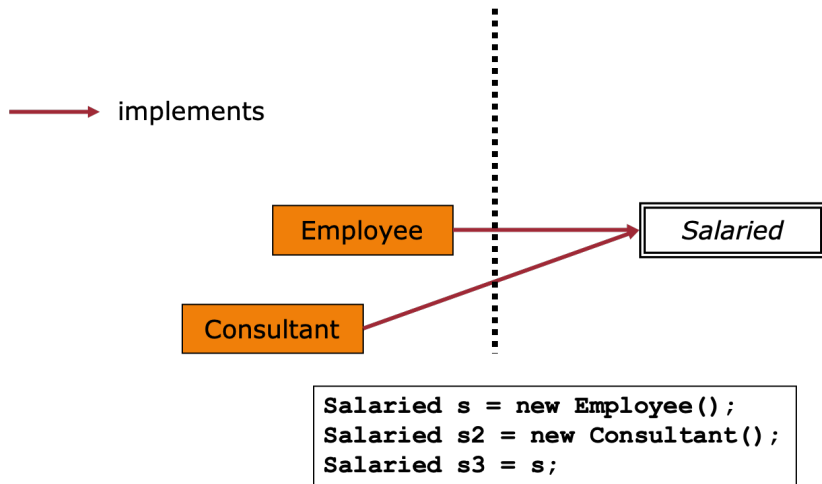
- ▶ Interfaces don't have constructors

```
payee = new Salaried(); //compile-time error
```

- ▶ Only *classes* can be instantiated directly
- ▶ Variable declared using the **interface** type can refer to an instance of a **class** that implements that interface

```
Salaried payee = new Employee(); //ok
```

Interfaces and Classes



Declared vs Dynamic Type

- ▶ **Declared type:** set at **compile** time (by declaration)
- ▶ **Dynamic type:** set at **run** time (by **new**)
`Type1 variable = new Type2();`

- ▶ **Examples:**

```
Employee p = new Employee("Pierre");  
Salaried s = new Employee("Liz", 12345);  
s = p; //dynamic type of s is: Employee
```

- ▶ Compiler can not infer dynamic type

```
void select(Salaried s) {  
    // declared type of s is: Salaried  
    // dynamic type of s is: ???  
    ...  
}
```

- ▶ Operator **instanceof** tests the run-time type

```
if (s instanceof Employee) { ... }  
else if (s instanceof Consultant) { ... }
```

Role of Declared Type

- ▶ Declared type determines which members can be used

```
class Employee implements Salaried {  
    public void setSalary(double d) { ... }  
    public double getSalary() { ... }  
    public void promote(int r) { ... }  
}
```

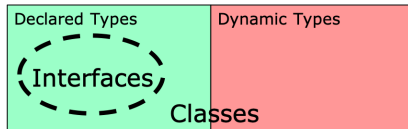
...

```
void select (Salaried s) {  
    s.setSalary(59000.00);  
    s.promote(0);    //compile-time error  
}
```

- ▶ Only *interface* members can be called/accessed by client
 - ▶ Class method is the code to execute when called
 - ▶ That method code can access all class members

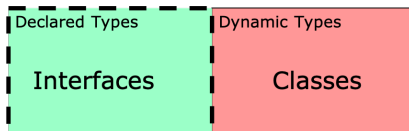
Simple Rule #1 - Compiler

- ▶ Rule: Interfaces can *only* be used as declared types
 - ▶ = Interfaces are never dynamic types
 - ▶ = Interfaces are never instantiated
 - ▶ = All dynamic types are classes
 - ▶ = All run-time objects are constructed from a class, not an interface



Good Practice: Code to Interface

- ▶ “Coding to the interface” means *all* declared types are **interface** types
 - ▶ All variable and field declarations use **interface** types
`Salaried lastHire = new Employee();`
 - ▶ All argument and return types in method signatures are **interface** types
`public Voter choose(Salaried[] s) { ... }`



Coding to the Interface

- ▶ Why?
- ▶ Code is written based only on the `interface`
 - ▶ No implementation details in the client code
 - ▶ Implementation can change and client code is not affected
 - ▶ Change implementation by changing a constructor call
- ▶ By using `interface` type as the argument type for methods, our methods are more flexible
 - ▶ Return types too
- ▶ **Example:** `List` vs. `ArrayList`

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 What is an interface?
- 2 Why interfaces?
- 3 How to develop and use interfaces in Java?