

Interfaces

Computer Science and Engineering • College of Engineering • The Ohio State University

Lecture 6

Syntax

Computer Science and Engineering • The Ohio State University

- An interface is a set of requirements
 - Describes *what* classes should do
 - Does not describe *how* they should do it
- Example

```
public interface Salaried {
    void setSalary(BigDecimal d);
    BigDecimal getSalary();
}
```
- To satisfy this interface, a class must provide *setSalary* and *getSalary* methods with
 - matching signatures (checked by compiler)
 - matching behaviors (up to you)

Good Practice: Use BigDecimal

Computer Science and Engineering • The Ohio State University

- ❑ Amounts of money (with pennies) should be represented with BigDecimal
 - `java.math.BigDecimal`
 - Methods for basic arithmetic operations
 - Rounds to given precision
 - Use `BigDecimal(String)` constructor, avoid `BigDecimal(double)`
- ❑ Double and float are always dangerous, due to rounding errors

```
System.out.println(4.56); //prints 4.56
System.out.println(4.56*100);
//prints 455.99999999999994
```

Declaring an Interface

Computer Science and Engineering • The Ohio State University

- ❑ 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)
 - ❑ These qualifiers usually omitted (implicit)
 - Methods
 - ❑ Must be **public abstract** (ie bodiless)
 - ❑ These qualifiers usually omitted (implicit)
 - ❑ Can not be **final** or **static**
- ❑ The interface itself is public or package visible

Examples

Computer Science and Engineering • The Ohio State University

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

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

Implementing an Interface

Computer Science and Engineering • The Ohio State University

- ❑ Declare a class that *implements* the interface
class Employee implements Salaried { . . . }
- ❑ Supply definitions for *all* interface methods
public void setSalary (BigDecimal d) {
 . . .
}
public BigDecimal 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

Eclipse Demo

Computer Science and Engineering ■ The Ohio State University

- See (interface) Salaried
 - Generate class (boiler plate) from interface
 - New > Class
 - Add interface Salaried
 - Make sure checkbox to create “inherited abstract methods” is selected
- See (class) SafePencil
 - Generate interface from class
 - Refactor > Extract Interface...
 - Select methods to include in interface
 - Problem: concrete representation driving the abstract view

Relationship with Resolve

Computer Science and Engineering ■ The Ohio State University

- Recall Resolve’s separation of client-side view and implementer’s view
- Client-side
 - Description of *what* a component does
 - Abstract state, the “mathematical model”
 - Requires and ensures clauses
- Implementer’s side
 - Description of *how* component works
 - Concrete state, the “representation”
- Matching concepts in Java
 - Interface: Client-side (“abstract instance/template”)
 - Class: Implementer (“concrete instance/template”)

Role of Interfaces vs Classes

Computer Science and Engineering • The Ohio State University

- Interfaces (should) provide
 - Method signatures
 - Mathematical model
 - Constraints (invariants on abstract state)
 - Method specifications
- Classes (should) provide
 - Concrete representation (in private fields)
 - Concrete implementation (in method bodies)
 - Conventions (invariants on concrete representation)
 - Correspondence (abstraction relation mapping concrete representation to abstract state)

Computer Science and Engineering • The Ohio State University

```
//Math Model: salary is a real number  
//Constraint (Abs Inv): salary >= 0  
public interface Salaried {  
  
    //Requires: d >= 0;  
    //Alters: this.salary  
    //Ensures: this.salary = d  
    void setSalary(BigDecimal d);  
  
    //Returns: this.salary  
    BigDecimal getSalary();  
}
```

Good Practice: Naming Interfaces

Computer Science and Engineering • The Ohio State University

- How should interfaces be distinguished from classes in their names?
- Resolve approach
 - Classes end in “_1” (or _2, _3,...)
 - eg `Pencil` vs `Pencil_1`
- Microsoft approach
 - Interfaces start with “I”
 - eg `IPencil` vs `Pencil`
- Java approach
 - No difference, both are nouns or adjectives
 - eg `WritingStick` vs `Pencil`

Instantiating an Interface

Computer Science and Engineering • The Ohio State University

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

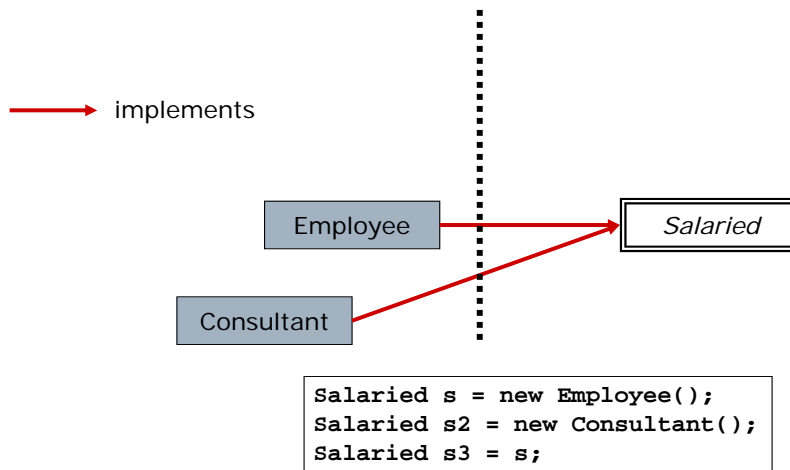
```
interface Salaried { . . . }
Salaried payee; //ok
```
- But interfaces cannot be instantiated directly

```
payee = new Salaried(); //compile-time error
```
- Only *classes* can be instantiated directly
- Variable of type *I* can refer to *an instance of a class that implements I*

```
class Employee implements Salaried { . . . }
Salaried payee = new Employee(); //ok
```
- (This might remind you of widening!)

Interfaces and Classes

Computer Science and Engineering • The Ohio State University



Declared vs Dynamic Type

Computer Science and Engineering • The Ohio State University

- 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:`
- 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

Computer Science and Engineering • The Ohio State University

- Declared type determines which members can be used

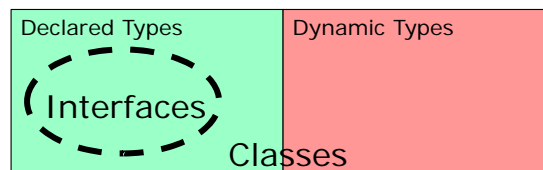
```
class Employee implements Salaried {
    public void setSalary (BigDecimal d) {...}
    public BigDecimal getSalary() {...}
    public void promote (int r) {...}
}
...
void select (Salaried s) {
    s.setSalary(new BigDecimal("59000.00"));
    s.promote(0); //compile-time error
}
```

- 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

Computer Science and Engineering • The Ohio State University

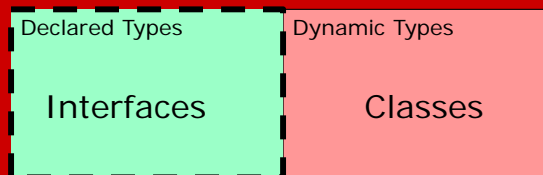
- Rule: Interfaces can *only* be used as declared types
 - = Interfaces are never dynamic types
 - = Interfaces are never instantiated
 - = All dynamic types are classes
 - = All run-time objects are constructed from a class, not an interface



Good Practice: Code to Interface

Computer Science and Engineering • The Ohio State University

- “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) {...}`



Implementing Multiple Interfaces

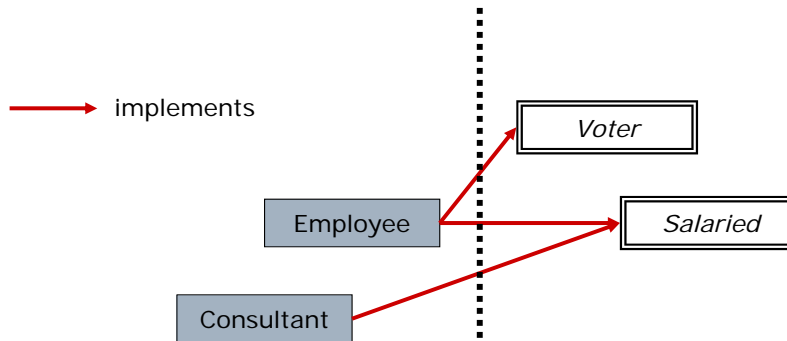
Computer Science and Engineering • The Ohio State University

- One class can implement several interfaces

```
class Employee implements Salaried,
    Voter {
    . . .
}
```
- Class must provide functionality from *all* interfaces it implements
 - Union of method signatures
 - Satisfies the behavioral contracts of all interfaces it implements

Multiple Interfaces

Computer Science and Engineering • The Ohio State University



```
Voter v = new Employee();
Salaried s = new Employee();
Salaried s2 = new Consultant();
Voter v2 = s; //compile-time error
```

Summary

Computer Science and Engineering • The Ohio State University

- Declaring an interface
 - Method signatures without implementation
 - Fields too, but this is less common
 - All implicitly public
- Implementing an interface
 - Class provides implementation for all methods
- Separation of client-side and implementation
 - Interface has abstract state, invariant, specs
 - Classes have concrete representation, convention
- Declared vs dynamic type
 - Interfaces can not be instantiated