

Interfaces

Lecture 5

Background

- 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)

Role of Interfaces vs Classes

- Interfaces (should) provide
 - Method signatures
 - Method specifications (in comments)
 - Mathematical model (in comments)
- Classes (should) provide
 - Concrete representation (in private fields)
 - Concrete implementation (in method bodies)
- Recall from RESOLVE the concepts of:
 - Representation invariant
 - Correspondence relation
 - Requires and ensures clauses
- All of these concepts can be projected onto Java interfaces and classes

Declaring an Interface

- Looks like a class definition, except:
 - Keyword *interface* replaces class
 - Methods have no body
- 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*
- Like a class, an interface can be generic

```
interface Holder<T> {
    void insert(T item);
    T remove();
}
```
- The interface itself is public or package

Implementing an Interface

- 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)
- One class can implement many interfaces
 - cf inheritance

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

Instantiating an Interface

- The static type of a variable can be an interface

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

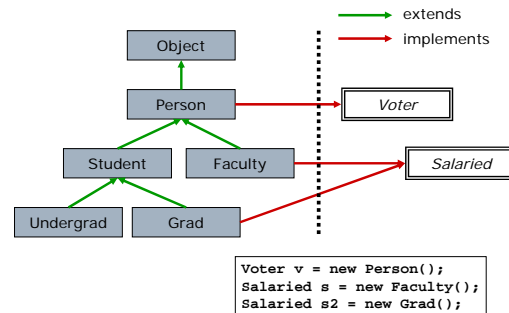
```
payee = new Salaried(); //compile-time error
```
- Only *classes* can be instantiated directly
- Variable of interface type I can refer to an instance of a class that implements I

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

Simple Rule #1

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

Class and Interface Hierarchies



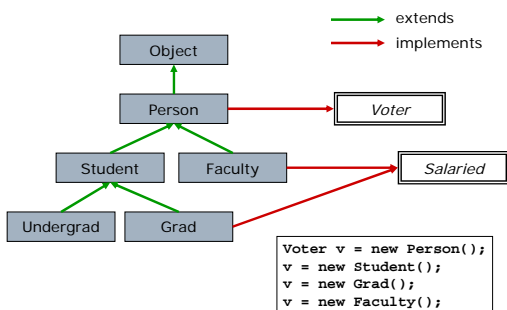
Best Practice: Code to Interface

- "Coding to the interface" means *all* static types are interface types
 - All variables declarations use interface types
 - `Salaried manager = new Employee();`
 - All argument and return types in method signatures are interface types
 - `public Voter choose(Salaried[] s) { . . . }`

Simple Rule #2

- A variable of static type T can refer to an object of dynamic type "at or below" T

Class and Interface Hierarchies

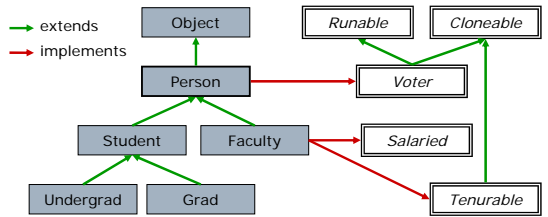


Extending Interfaces

- Interfaces can inherit from other interfaces
- Vocabulary:
 - subinterface/superinterface
- Multiple inheritance is allowed
 - cf single inheritance for classes
- No (implicit) base interface for all interfaces
 - cf `java.lang.Object` for classes
- Syntax similar to class inheritance


```
interface X extends A, B { . . . }
```

Class and Interface Hierarchies



Faculty extends Person, Object
Faculty implements Salaried, Tenurable, Voter, Runnable, Cloneable

Best Practice: Which Static Type?

- How specific should the static type of an argument / return type be?

```

Voter selectRepresentative(Tenurable[] A)
Runnable selectRepresentative(Tenurable[] A)
Voter selectRepresentative(Cloneable[] A)
Runnable selectRepresentative(Cloneable[] A)
  
```

- Typical advice:
 - "As specific as possible, without revealing implementation details"
 - "As general as possible, while still being useful to client"
- The right way to think about it:
 - The type is dictated by the mathematical (abstract, client-side) model

Abstract Classes

- A class can be declared to be *abstract*

```

abstract class Design { . . . }
  
```

 - Can not be instantiated (same as interfaces)
 - May contain *abstract methods*
- An *abstract method* has no implementation


```

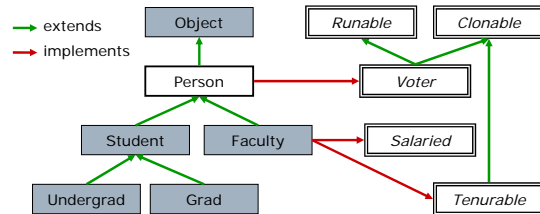
abstract class Design {
    void setLabel() { . . . }
    abstract int getCost();
}
  
```
- Only a subclass that implements all of these abstract methods can be instantiated


```

class Drawing extends Design {
    int getCost() { . . . }
}
  
```

 - Otherwise, the subclass is abstract too
- Combination of interface and class

Class and Interface Hierarchies



Instantiable?
 Yes: Object, Student, Faculty, Undergrad, Grad
 No: Person, Salaried, Tenurable, Voter, Runnable, Cloneable

Java Support for Subtyping

- Java does not enforce behavioral contracts
 - Type checking ensures only that arguments match
- Support for behavioral subtyping limited to very weak promises, such as:
 - If B has a visible method $m()$, A has a visible method $m()$ with same signature
 - A can not *decrease* visibility of $m()$!
 - Arguments must match exactly
 - Return type *can be* a subtype (covariance)
 - If B's method $m()$ can not throw an exception of type E, neither can A's $m()$
 - A can not *increase* the list of possible exceptions
 - We'll talk about exceptions later...

Best Practice: Limited Use

- Getting *implementation inheritance* right is hard
 - Interface* inheritance is much easier to get right
- Unless you have an explicit *need* for an open (ie extendable) class hierarchy, prevent others from extending your classes
- Keyword *final* prevents extensions


```

public final class Faculty {
    . . .
}

public class Administrator extends Faculty {
    . . . //compiler complains
}
  
```
- If you *do* have a specific need to allow extensions, design for it carefully
 - Use protected diligently and carefully (it's a huge increase in visibility over private or even over package!)
 - Chances are, it will still be broken