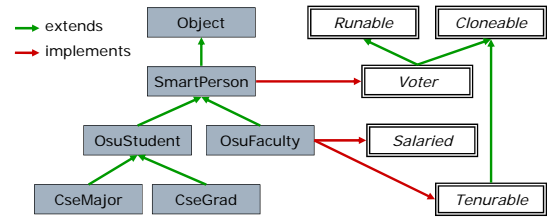


Inheritance: Applications and Consequences

Lecture 13

Class and Interface Hierarchies



OsuFaculty extends SmartPerson, Object
 OsuFaculty implements Salaried, Tenurable, Voter, Runnable, Cloneable

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


```

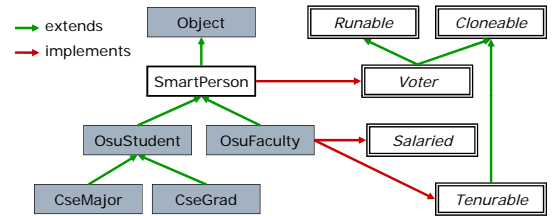
abstract class Design {
    void setLabel() { . . . }
    abstract int getCost();
}
      
```
- Only a subclass that implements all of these abstract methods can be instantiated


```

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

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

Class and Interface Hierarchies



Instantiable?
 Yes: Object, OsuStudent, OsuFaculty, CseMajor, CseGrad
 No: SmartPerson, Runnable, Cloneable, Voter, Salaried, Tenurable

Abstract Classes vs. Interfaces

- Similarities
 - Neither can be instantiated
- Differences
 - Abstract classes permit:
 - Constructors
 - Static methods
 - Fields (but these are not part of public interface anyway, right?)
 - Visibilities: private/protected/default/public
 - Implementations
 - Interfaces permit:
 - Multiple inheritance

Controlling Inheritance: final

- Ultimate control: disallow
 - Declare class to be final


```
final class CseMajor { ... }
```
 - Abstract classes can not be final


```
final abstract class SmartPerson { ... }
```
- Finer granularity: Disallow certain methods to be overridden
 - Declare method to be final


```
abstract class SmartPerson {
    final int getAge() { . . . }
}
```
 - Permitted in abstract classes, but an abstract method can not be final
 - cf C++ (explicitly permit overriding with virtual)

Hook and Template Methods

- Recall pattern:
 - Base class contains both template and hook methods
 - Template method calls this.hook method
 - Hook methods are overridden in derived classes
 - Template method is not
- To support this pattern:
 - Template method is declared final
 - Hook methods are declared abstract
 - So base class declared abstract too
 - Hook methods are declared protected
- See divide-and-conquer example
 - solve() is the template method

Hook and Template Idiom

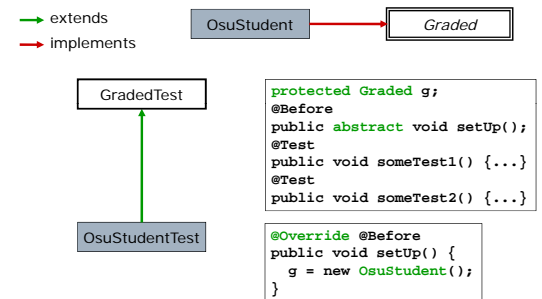
```
public abstract class Course {
    public final void enroll(Student s) {
        if (checkEligibility(s)) { ... }
    }
    protected abstract boolean
        checkEligibility(Student s);
}

public class Tutorial extends Course {
    @Override
    protected boolean
        checkEligibility(Student s) {
        //determines whether s has paid
    }
}
```

JUnit Pattern

- Goal: Separate interface and implementation tests
 - Former are based on abstract client-side view
 - Latter based on concrete implementer's view
- Approach:
 - Test fixture for interface tests is a base class
 - Test fixture for implementation tests extends it
- JUnit tests require an object (class instance)
 - In base class:
 - Use protected member(s) of interface type
 - abstract @Before method
 - In derived class:
 - Override @Before method to instantiate class and initialize the protected member(s)
- See RandomWithParity example

JUnit with Inheritance



Limitations of This JUnit Pattern

- Limitation 1: Single inheritance
 - If interface A extends B, no problem: test fixture ATest simply extends test fixture BTest!
 - But interface A extends B, C is trouble
 - Reason: with classes we are limited to single inheritance
- Limitation 2: Complex construction
 - Assumes test cases do not require a particular constructor call for the class under test (all use default constructor)
 - What if this is not the case? (eg BigNatural)
 - Solution: Factory methods (We'll see these later)

Javadoc

- Javadoc comments (main description, @param, @return) are implicitly inherited when omitted for a method
 - In a class that overrides a method in superclass
 - In an interface that overrides a method in superinterface
 - In a class that implements a method in interface
- Javadoc generates "Overrides" block for first two, and "Specified by" block for last one
 - Links to comment for that parent method
- {@inheritDoc} explicitly inherits parent's comment
 - Replaced by text of parent's comment (can augment)
 - Use in main description, @param, @return

Narrowing

- Recall that narrowing requires explicit cast
 - Programmer promise that this is OK

```
void v(OsuStudent s) {
    ((CseMajor)s).assignJavaLab();
}
```
- What if the programmer is wrong?
 - Results in run-time failure (an "exception")
- Programmer can check first if it is OK
 - Operator: *instanceof*

```
if (v instanceof BankAccount) {
    ((BankAccount)v).deposit();
    . . .
}
```
- Beware:
 - Any use of *instanceof* in code is a **red flag**
 - Especially bad smell: *switch()* based on *instanceof*

Surprise?

- Static methods are inherited
- But, they do *not* get polymorphic run-time selection
 - Implementation selected according to *declared type*
 - Yet another reason to invoke static methods through class (not an instance)

To Ponder:

```
public class Base {
    public static int f() {
        return 4;
    }
}
public class Derived extends Base {
    public static int f() {
        return 8;
    }
}
...
Base b = new Derived();
System.out.println(b.f());
//What does this print?
```

Good Practice: Static Members

- Do *not* access static members through object references
- Use class names instead
 - Do this: `int t = Pencil.defaultLength;`
 - Not this: `int t = pl.defaultLength;`
- This applies within a class too

```
class Pencil {
    private static int defaultLength = 10;
    private int length;
    public void reset() {
        length = defaultLength; //correct
        length = Pencil.defaultLength; //better
    }
}
```

To Ponder:

```
public class Base {
    public static int f() {
        return 4;
    }
}
public class Derived extends Base {
    public static int f() {
        return 8;
    }
}
...
System.out.println(Base.f());
System.out.println(Derived.f());
//What does this print?
```

Inheritance Myths

- class A extends B implies A is a behavioral subtype of B
- No! Overriding methods could break everything

Inheritance Myths

Computer Science and Engineering @ The Ohio State University

- ❑ If I don't override any methods, everything is fine
- ❑ No! Adding new methods could break the invariant!

Summary

Computer Science and Engineering @ The Ohio State University

- ❑ Abstract classes
 - Contain abstract methods
 - Missing some implementation
 - Like interfaces, can not be instantiated
- ❑ Final methods
 - Can prevent overriding specific methods
- ❑ Template and hook pattern
 - Template class and hook methods all abstract
 - Template method is final
- ❑ Leveraging inheritance for JUnit
- ❑ Javadoc features
- ❑ Static methods can not be overridden
- ❑ Inheritance myths