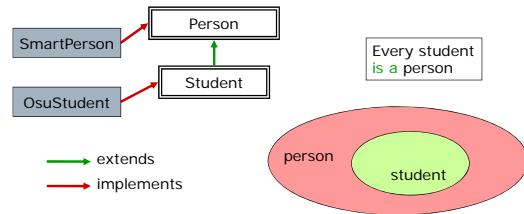


Implementation Inheritance

Lecture 12

Recall: Interface Inheritance

```
void select (Person p) {
    //declared type of p is:
    //dynamic type of p is:
```



Recall: Behavioral Subtyping

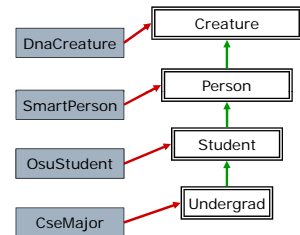
- A Student can do everything a Person can do
- Everywhere a Person is expected, a Student can be used instead

```
void select (Person p) {
    if (p.getAge() > 18) {
        p.summons(trialDate);
        ... etc ...
```

- Every method promised in Person interface:
 - Is implemented in SmartPerson class
 - Is promised in Student interface
 - Is implemented in OsuStudent class
- Are two separate implementations of getAge really necessary (or even a good idea)?

More Extreme Example

- Every method promised in Creature interface:
 - Also promised in Person, Student, and Undergrad interfaces
 - Must be implemented in DnaCreature, SmartPerson, OsuStudent, and CseMajor classes!

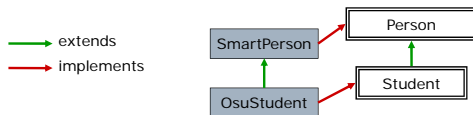


Implementation Inheritance

- Keyword: extends

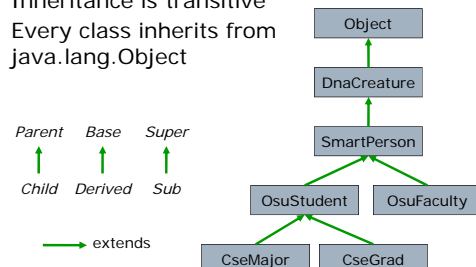

```
public class OsuStudent extends SmartPerson {
    ...
}
```

 - OsuStudent has SmartPerson's members (fields + methods, including *implementation*)
 - If omitted, java.lang.Object is implicit

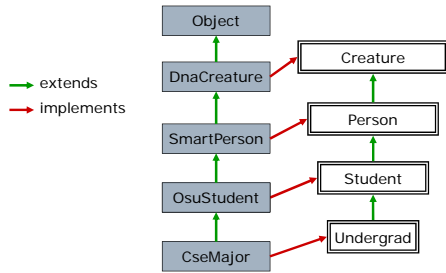


Class Hierarchy

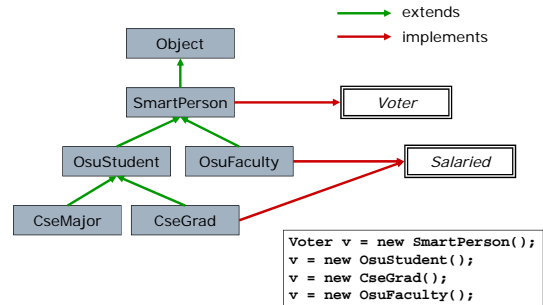
- Inheritance is transitive
- Every class inherits from java.lang.Object



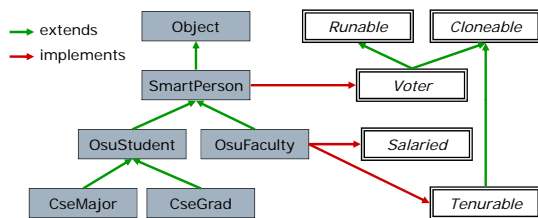
Class and Interface Hierarchies



Class and Interface Hierarchies



Class and Interface Hierarchies



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

Mechanics

- A class extends *exactly one* other class
 - "single inheritance" (cf C++)
- A subclass inherits all the members of its superclass
 - Does not have access to the private members
 - Does not inherit the constructors (ie just fields and methods)
- Subclass can add new members (hence "extends")
 - New fields and new methods
 - Defines its own constructor(s)
- Subclass can modify inherited methods
 - Changes behavior
 - "overriding"

Example: Code

```

class SmartPerson implements
    Person {
    private String name;

    SmartPerson() {
        name = "Baby Doe";
    }

    SmartPerson(String name) {
        this.name = name;
    }

    void rename(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }
}

class OsuStudent implements
    Student extends SmartPerson {
    private int identity;

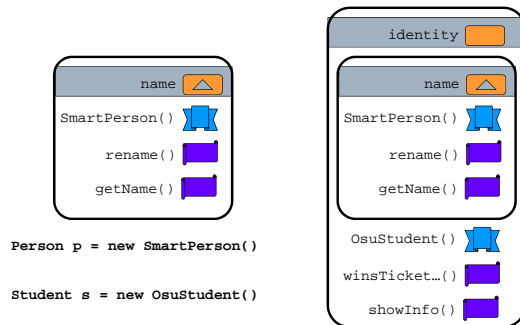
    OsuStudent() {
        identity = 0;
    }

    OsuStudent(String name, int
        identity) {
        super(name);
        this.identity = identity;
    }

    boolean winsTicketLottery () {
        return (identity % 13 == 0);
    }

    String showInfo () {
        return "[" + getName() +
            identity + "]";
    }
}
    
```

Example: Graphical View



Constructing New Instances

- Members of OsuStudent:
 - Its own: identity, winsTicketLottery(), showInfo()
 - Its parent's: rename(), getName()
 - Its parent's parent's: see java.lang.Object
 - eg clone(), equals(), hashCode(),...
- When a new instance is created:
 - First, the parent's constructor is invoked
 - Can be done explicitly with super()
 - Otherwise, parent's default constructor is called
 - Next, any initialization blocks are executed
 - Finally, the child's constructor is executed

Overriding Methods

- Overriding:** a subclass declares a method that is already present in its superclass
- Note: signatures must match (otherwise it is just overloading)

```
class SmartPerson {
    String showInfo() {
        return getName();
    }
}
class OsuStudent extends SmartPerson {
    String showInfo() {
        return "[" + getName() + identity + "]";
    }
}
```
- Question: which method is called?

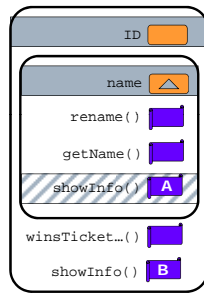
```
SmartPerson p = new OsuStudent();
System.out.println(p.showInfo());
```

 - Declared type: SmartPerson, dynamic type: OsuStudent

Overriding: Graphical View

```
OsuStudent s = new OsuStudent()
s.showInfo(); //impl: B

SmartPerson p = s;
p.winsTicketLottery(); //error
p.showInfo(); //impl: A or B?
```



Polymorphism

- Answer: The *dynamic type* determines which method is called

```
SmartPerson p = new OsuStudent();
p.showInfo() //calls OsuStudent version
```
- Informal model:
 - Method invocation is a run-time message to the object
 - That (run-time) object receives the request, performs the action, and returns the result
- Goal: we get the right behavior regardless of actual (ie run-time, ie dynamic) type

```
Person[] csePeople = ... //students & faculty in CSE
for (int i = 0; i < csePeople.length; i++) {
    ...csePeople[i].showInfo()...;
}
```
- Note: This applies to methods only, not fields
 - Fields can not be overridden, only hidden

Good Practice: @Override

- Use **@Override** annotation with all methods intended to override a method in a superclass

```
class OsuStudent extends SmartPerson {
    @Override
    String getInfo() {
        ...
    }
}
```
- Compiler complains if there is no matching method in superclass
 - Prevents accidental overloading if a mistake is made in the signature
- Beware: Differences between Java 5 & 6

Hook methods

- Dynamic type of *this* controls which method executes
- Hook method: Called internally, intended to be overridden

```
class Course {
    void enroll(Student s) {
        if (this.checkEligibility(s)) { ... }
    }
    boolean checkEligibility(Student s) {
        //determines whether s has prereqs for this course
    }
}
class Tutorial extends Course {
    boolean checkEligibility(Student s) {
        //determines whether s has paid fees
    }
}
```
- Yo-yo problem:
 - Must trace up & down class hierarchy to understand code

```
Course workshop = new Tutorial();
workshop.enroll(s);
```

Protected

Computer Science and Engineering @ The Ohio State University

- We have seen three levels of visibility
 - private: concrete representation
 - default (ie package): trusted and co-located
 - public: abstract interface to all clients
- Writing a subclass may require:
 - More access than client-view (abstract interface)
 - Less access than whole concrete representation
- Solution: new visibility level
 - Keyword: *protected*
 - Protected members *are* inherited but are *not* part of the public interface to generic clients
 - Warning: anyone can extend your class and then has access to protected members

Good Practice: Limited Use

Computer Science and Engineering @ The Ohio State University

- Getting it right is hard
- 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

To Ponder

Computer Science and Engineering @ The Ohio State University

```
class Course {
    public int enrollment() {
        return 24;
    }
    . . .
    void f(Course c) {
        System.out.println(c.enrollment());
        //What does this print?
    }
}
```

Summary

Computer Science and Engineering @ The Ohio State University

- Implementation (class) inheritance
 - Declaration syntax: extends just like interfaces
 - Vocabulary: super/sub, base/derived, parent/child
- Class and interface hierarchies
 - Constructing new instances
- Overriding and polymorphism
 - Signature must match exactly (use @Override)
 - Dynamic type controls implementation
 - Hook methods: dynamic type of this
- Protected visibility
- Limiting extension: final