Implementation Inheritance

Recall: Interface Inheritance

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

Person

SmartPerson

Student

OsuStudent

Every student is a person

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(Date 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
  - Implemented in DnaCreature, SmartPerson, OsuStudent, and CseMajor!

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
- Vocabulary
  - superclass/subclass
  - base class/derived class
  - parent class/child class

Object

DnaCreature

SmartPerson

OsuStudent

OsuFaculty

CseMajor

CseGrad

Student

Person

SmartPerson

OsuStudent

DnaCreature

SmartPerson

OsuStudent

CseMajor
Mechanics
- Can extend exactly one other class
  - "single inheritance" (cf C++)
- A subclass has all the members of its superclass!
  - Not the private members
  - Not 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
```java
class SmartPerson {
    private String name;
    SmartPerson () {
        this.name = "Baby Doe";
    }
    SmartPerson (String name) {
        this.name = name;
    }
    void rename (String name) {
        this.name = name;
    }
    String getName () {
        return this.name;
    }
}
class OsuStudent extends SmartPerson {
    private int ID;
    OsuStudent () {
        this.ID = 0;
    }
    OsuStudent (String name, int ID) {
        super(name);
        this.ID = ID;
    }
    boolean winsTicketLottery () {
        return (this.ID % 2 == 0);
    }
    String toString () {
        return this.getName() + " [" + this.ID + "]";
    }
}
```

Example: Graphical View
```
SmartPerson p = new SmartPerson()
OsuStudent s = new OsuStudent()

Example: Constructing New Instances
- Members of OsuStudent:
  - its own: ID, winsTicketLottery(), toString()
  - 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 toString() {
        return this.getName();
    }
}

class OsuStudent extends SmartPerson {
    String toString() {
        return this.getName() + " [" + this.ID + "]";
    }
}
```

- Question: which method is called?

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

Static type is SmartPerson, dynamic type is OsuStudent

Overriding: Graphical View

```
OsuStudent s = new OsuStudent();
s.toString(); //impl: B
SmartPerson p = s
p.toString(); //impl: B
```

Polymorphism

- Answer: The dynamic type determines which method is called

```
SmartPerson p = new OsuStudent();
p.toString(); //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 (run-time) type

```
Person[] csePeople = … //students & faculty in CSE
for (int i = 0; i < csePeople[].length; i++) {
    csePeople[i].toString();
}
```

- 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 getName() {
        …
    }
}
```

- Compiler complains if there is no matching method in superclass
  - Prevents accidental overloading if a mistake is made in the signature
- We (may) talk about annotations later...

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 (checkEligibility(s)) { … }
    }
}

class Tutorial extends Course {
    boolean checkEligibility(Student s) {
        // determines whether s has prerequisites for this course
        …
    }
}
```

- Yo-yo problem:
  - Must trace up & down class hierarchy to understand code

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

Protected

- We’ve seen three levels of visibility
  - private: concrete representation
  - default: trusted and collocated
  - public: abstract interface to all clients
- Writing a subclass often requires:
  - More access than a generic client
  - 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 get access to protected members
Good Practice: Limited Use

- Getting it right is hard
- Unless you have an explicit need for an open (i.e., extendable) class hierarchy, prevent others from extending your classes
- Keyword final prevents extensions
  ```java
  public final class Faculty {
  . . .
  }
  ``
  ```java
  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

Summary

- 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