CIS 110: Introduction to computer programming

Lecture 25

Inheritance and polymorphism

(§ 9)

Outline

- Inheritance
- Polymorphism
- Interfaces

Inheritance

Example: student and faculty records

- Consider parsing a file of student/faculty records.

```
alur,faculty,35,Levine 609,professor
susan,student,18,3.9,junior
zives,faculty,32,Levine
566,associate
lee,student,20,3.5,senior
nenkova,student,21,4.0,freshman
```


The Student class

```
public class Student {
  private String username;
  private int age;
  private double gpa;
  private String status;

  public Student(String username, int age, double gpa, String status) {
    this.username = username;
    this.age = age;
    this.gpa = gpa;
    this.status = status;
  }

  public String getUsername() { return username; }
  public int getAge() { return age; }
  public double getGPA()  { return gpa; }
  public String getStatus() { return status; }
}
```


The Faculty class

```
public class Faculty {
  private String username;
  private int age;
  private String office;
  private String status;

  public Faculty(String username, int age, String office, String status) {
    this.username = username;
    this.age = age;
    this.office = office;
    this.status = status;
  }

  public String getUsername() { return username; }
  public int getAge() { return age; }
  public String getOffice() { return office; }
  public String getStatus() { return status; }
}
```

Class redundancy

- Student and faculty share fields and methods!
  - username/age, getUsername, getAge
- How do we factor out class redundancy?
  - Insight: a student and a faculty are related somehow…

The Person class

```
public class Employee {
  // For simplicity's sake, let's just consider methods...
  public int getAge() { return 20; }
  public String getUsername() { return "username"; }
}
```


Extending classes - inheritance

- We can *extend* the Person class to inherit its two methods.

```
public class Student extends Employee { /** no impl yet */ }
```

- Now we can call methods of the Employee class on Student objects.

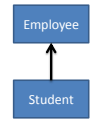
```
Student s = /* ... */;
s.getUsername();
s.getAge();
```

A class can only extend at most one other class!

Inheritance of terminology

```
public class Employee { /* ... */ }
public class Student extends Employee { /* ... */ }
```

- We say that
  - Student *extends* Employee.
  - Student *inherits* Employee.
  - Student *derives from* Employee.
  - Student is a *subclass* of Employee.
  - Employee is the *superclass* of Student.
  - Employee is the *parent class* of Student.
  - Student is-a Employee.



Inheritance hierarchy

Adding onto the Student class

```
public class Student extends Employee {
  public double getGPA()  { return 4.0; }
  public String getStatus() { return "Junior"; }
}
```

- We can call four methods on a Student object.
  - Two from Employee (getUsername(), getAge()).
  - Two from Student (getGPA(), getStatus()).
- Inheritance allows for *code sharing* between classes.
  - Only one of the benefits!

Inheriting state as well as behavior

- Let's add state back into the classes:

```
public class Employee {
  private String username;
  private int age;
  public Employee(String username,
                  int age) {
    this.username = username;
    this.age = age;
  }
  public int getAge() { return age; }
  public String getUsername() {
    return username;
  }
}
```

```
public class Student extends Employee {
  private double gpa;
  private String status;
  public Student(double gpa,
                 String status) {
    this.gpa = gpa;
    this.status = status;
  }
  public double getGPA() { return gpa; }
  public String getStatus() {
    return status;
  }
}
```

BAD!! DOES NOT COMPILE!

Setting up your superclass

```
public class Student extends Employee {
  private double gpa;
  private String status;
  public Student(String username, int age, double gpa, String status) {
    super(username, age);
    this.gpa = gpa;
    this.status = status;
  }
  public double getGPA()  { return gpa; }
  public String getStatus() { return status; }
}
```

- We "set up our parent" by calling its constructor with super(…).

Accessing inherited members

- Say I want to allow students to change their age (but not faculty).

```
public class Student extends Employee {
  // ...
  public void setAge(int age) { this.age = age; }
}
```

BAD!! DOES NOT COMPILE!

- Employee's age field is marked private and thus is not visible from Student.

The protected modifier

- `public` = visible to everyone
- `private` = visible to only me (the class)
- `protected` = visible to me + all my subclasses

```
public class Employee {
  // ...
  protected int age;
}
```

- Allows flexibility at the cost of encapsulation…

Overriding methods

- Say we want to prepend the username with the status of the student.
- Solution: let's *override* the behavior of getUsername in Student.

```
public class Employee {
  // ...
  protected String username;
}

public class Student extends Employee {
  // ...
  public String getUsername() { return status + ":" + username; }
}
```


Invoking superclass methods

- However, say we don't want to expose write access to username to Student.
- Instead, let's invoke Employee's getUsername method directly instead of making username protected!

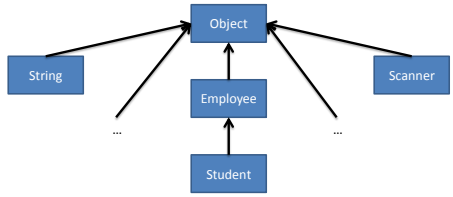
```
public class Employee {
  // ...
  private String username;
}

public class Student extends Employee {
  // ...
  public String getUsername() {
    return super.getUsername() + ":" + username;
  }
}
```

"super" = invoke my parent's version of this method.

The Object class

- Object is the ultimate superclass for all other Java classes.



Important methods of the object class

```
// Returns the String representation of this object
public String toString();
// Returns true if this object is equal to other
public boolean equals(Object other);
```

```
public class Employee {
  public boolean equals(Object other) {
    if (other instanceof Employee) {
      Employee e = (Employee) other;
      return username.equals(e.username) &&
             age == e.age;
    } else {
      return false;
    }
  }
}
```

`instanceof` = binary operator that is true if other is the same class or a subclass of Employee

Polymorphism

Student is-a Employee

- When Student extends from Employee, we say Student **is-a** employee:
  - Student is a specialization of Employee.
    - Student has all the behavior and potentially more!
  - **Student has all of the functionality of Employee.**

Polymorphism

- Because a Student is-a Employee, this works!
  - Polymorphism: "many forms", the same code can be used with many types.

```
Employee e = new Student(...);
```

The *static type* of e, i.e., the type e is declared with

The *dynamic type* of e, i.e., the actual type of the object.

- Intuition: Student does everything Employee can do (by virtue of extends) so we can use a Student where ever an Employee is expected.

Method calls and polymorphism

- What gets returned for these method calls?

```
public class Employee {
  public String toString() {
    return "Employee: " +
           username;
  }
}
```

```
public class Student
    extends Employee {
  public String toString() {
    return "Student: " +
           getUsername();
  }
}
```

```
Employee e1 = new Employee(...);
Student s1 = new Student(...);
Employee e2 = new Employee(...);
e1.toString();
s1.toString();
e2.toString();
```

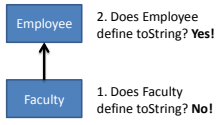
```
"Employee: ..."
"Student: ..."
"Student: ..."
```


Dynamic dispatch

- When resolving a method call, we start with the *dynamic* (actual) type of the object.
  - If that class defines the method, we invoke it.
  - Otherwise, we repeat the process with its immediate superclass.

```
Employee e = new Faculty(...);
e.toString();
```

```
"Employee: ..."
```



2. Does Employee define toString? **Yes!**

1. Does Faculty define toString? **No!**

To inherit or not to inherit

- Inheritance should be used when one class can be substituted for another.
  - Really, class A **is-a** B → `class A extends B`.
  - E.g., a Circle **IS-NOT** a Point even though a circle is a point + a radius.
- Alternative: **has-a** relationship.
  - A Circle **has-a** Point as a field.

Interfaces

`extends` isn't enough

- Sometimes we want to be able to have two or more is-a relationships for a class.
  - e.g., employees that are comparable in addition to being people.
- Sometimes we don't need to share code.
  - Only need to specify a set of "required" methods.

Introducing interfaces

- Interfaces allow us to specify the requirements for an **is-a** relationship without providing any implementation details.

```
public interface Comparable {
  public int compareTo(Object other);
}
```

```
public class Student extends Employee implements Comparable {
  // ...
  public int compareTo(Object other) { /* ... */ }
}
```


The interface construct

```
public interface Comparable {
  public int compareTo(Object other);
}
```

- Interfaces specify a series of abstract methods that a class must implement to, e.g., be Comparable.
- Classes can implement multiple interfaces (but only extend from a single class).
- Allows us to get polymorphism without code sharing.