Chap 7 - Inheritance

An example from Zoology

An example from java.lang

Other terms from inheritance

//Person.java - characteristics common to all people
class Person {
    String name;
    int age;
    char gender; // male == 'M', female == 'F'

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

    void setAge(int age) {
        this.age = age;
    }

    void setGender(char gender) {
        this.gender = gender;
    }

    int getAge() {
        return age;
    }

    char getGender() {
        return gender;
    }

    String getName() {
        return name;
    }

    public String toString() {
        return "Name: " + name + 
              ", Age: " + age + 
              ", Gender: " + gender;
    }

    // methods
}

//Student.java - an example subclass
class Student extends Person {
    String college;
    double gpa;
    byte year;

    Student(String name) {
        super(name);
    }

    void setCollege(String college) {
        this.college = college;
    }

    void setGpa(double gpa) {
        this.gpa = gpa;
    }

    void setYear(byte year) {
        this.year = year;
    }

    String getCollege() {
        return college;
    }

    double getGpa() {
        return gpa;
    }

    byte getYear() {
        return year;
    }

    public String toString() {
        return super.toString() + 
              "College: " + college + 
              ", GPA: " + gpa + 
              ", Year: " + year;
    }
}

//Student.java - an example subclass

static final byte FROSH = 1;
static final byte SOPH = 2;
static final byte JUNIOR = 3;
static final byte SENIOR = 4;
private String college = "Unknown";
private byte year; // FROSH, SOPH, ...
private double gpa; //0.0 to 4.0

//StudentTest.java
class StudentTest {
    public static void main(String[] args) {
        Student student = new Student("Jane Programmer");
        student.setAge(21);
        student.setGender('F');
        student.setCollege("UCSC");
        student.setYear(Student.FROSH);
        student.setGpa(3.75f);
        System.out.println(student.toString());
    }
}

Subtype Principle

• Methods defined in one class may be redefined in a subclass. This is called method overriding.
• A subclass object can always be used where an object of its superclass is expected.
• Treating a subclass object as a superclass object can only remove capabilities, not add them. With inheritance, new methods are added in the subclass, never taken away.

Can't use Student fields from Person reference

class StudentTest {
    public static void main(String[] args) {
        Student student = new Student("Jane Programmer");
        student.setAge(21);
        ...
        System.out.println(student.toString());
        Person anyPerson;
        if (Console.in.readInt() == 1)
            anyPerson = student;
        else
            anyPerson = new Person("John Doe");
        anyPerson.setYear(Student.FROSH); // illegal
        System.out.println(anyPerson.toString());
    }
}

Overriding vs Overloading

• Overloading is when you define two methods with the same name, in the same class, distinguished by their signatures.
• Overriding is when you redefine a method that has already been defined in a parent class (using the exact same signature).
• Overloading is resolved at compile time.
• Overriding is resolved at runtime (based on the type of the implicit first parameter).
Dynamic Method Dispatch

- Determining at runtime, which overridden method to call, is called dynamic method dispatch.
- This is what allows println() to work with any object.
- toString() is defined in Object (the parent of all classes).
- If you override toString(), then inside of println(), a call to printThis.toString(), will get to YOUR toString().

```java
public class PrintWriter {
    ...
    public void println(Object obj) {
        String s = obj.toString();
        // somehow get the string s printed
    }
}
```

```java
// SuperClass.java - a sample super class
class SuperClass {
    public void print() {
        System.out.println("inside SuperClass");
    }
}

// SubClass.java - a subclass of SuperClass
class SubClass extends SuperClass {
    public void print() {
        System.out.println("inside SubClass");
    }
}
```

Dynamic Method Dispatch

```java
// TestInherit.java - overridden method selection.
class TestInherit {
    public static void main(String[] args) {
        SuperClass s = new SuperClass();
        s.print();
        s = new SubClass();
        s.print();
    }
}
```

Access Modifiers, Method Overriding and the Subtype Principle

- When you override a method you cannot restrict access more than was done with the inherited method.
- Doing so would break the subtype principle.
- For example, if class Student override toString() to be private, then
  System.out.println(anyPerson.toString());
  would fail if anyPerson was referring to a Student.

```java
// Generic Methods
• A generic method is a method that can operate on at least two different types of data. This is a form of polymorphism.
• println(Object obj) is a generic method. It works for any Java reference type.
• In addition to toString(), the class Object defines a method
  public boolean equals(Object obj);
```
Method equals() in Object

- By overriding equals() we can write a generic methods that need to check if two objects are equal.

```java
// Count the number of times obj is found in array.
static int howManyCopiesOf(Object obj, Object[] array) {
    int count = 0;
    for (int i = 0; i < array.length; i++)
        if (obj.equals(array[i]))
            count++;
    return count;
}
```

class EqualsTest {
    public static void main(String[] args) {
        // Create and fill an array of Strings
        String[] stringArray = new String[10];
        for (int i = 0; i < stringArray.length; i++)
            stringArray[i] = "String " + i;
        // Create and fill an array of Counters
        Counter[] counterArray = new Counter[5];
        for (int i = 0; i < counterArray.length; i++)
            counterArray[i] = new Counter();
        // Make two entries refer to the same Counter
        counterArray[2] = counterArray[0];
        System.out.println(
            howManyCopiesOf(counterArray[0], counterArray));
        System.out.println(
            howManyCopiesOf("String 1", stringArray));
    }
}

When equals are not equal?

- Using the original Counter from chapter 6, the output of the previous program is 2 and 1. That's 2 Counters "equal" to counterArray[0], and 1 String "equal" to "String 1".
- The class String, overrides equals() to compare the characters in two Strings.
- Counter uses the default equals() which just checks if the references point to the same object.

Primitive Wrapper Classes

- Object
- Number
  - Long
  - Integer
  - Double
  - Float
A Generic Numeric Method

```java
static Number elementMin(Number[] array) {
    Number min = array[0];
    for (int i = 1; i < array.length; i++)
        if (array[i].doubleValue() < min.doubleValue())
            min = array[i];
    return min;
}
```

This works so long as the loss of precision in converting to double doesn’t affect the selected minimum.

Abstract Classes

- An abstract class is used to derive other (concrete) classes.
- An abstract class usually provides some complete method definitions, but leaves some undefined, requiring all subclasses to provide definitions for the undefined methods.

```java
abstract public class AbstractCounter {
    abstract public void click();
    public int get() { return value; }
    public void set(int x) { value = x; }
    public String toString()
    { return String.valueOf(value); }
    protected int value;
}
```

```java
public class Counter extends AbstractCounter {
    public void click() { value = (value + 1) % 100; }
}
```

```java
public class CountByTwo extends AbstractCounter {
    public void click() { value = (value + 2) % 100; }
}
```

```java
public class Timer extends AbstractCounter {
    public Timer(int v) { set(v); }
    public void click() {
        value++;  // time incremented
        seconds = value % 60;
        minutes = value / 60;
    }
    public void set(int v) {
        value = v;
        seconds = v % 60;
        minutes = v / 60;
    }
    public String toString() {
        return minutes + " minutes, " +
                seconds + " seconds";
    }
    private int seconds, minutes;
}
```

Predator-Prey: An abstract class

A simple simulation of an artificial ecology

```java
class Predator extends AbstractCounter {
    public Predator(int v) { set(v); }
    public void click() {
        value++;  // time incremented
        seconds = value % 60;
        minutes = value / 60;
    }
    public void set(int v) {
        value = v;
        seconds = v % 60;
        minutes = v / 60;
    }
    public String toString() {
        return minutes + " minutes, " +
                seconds + " seconds";
    }
    private int seconds, minutes;
}
```
### Pseudocode for Predator-Prey

- create two worlds, current and next
- initialize one with some life forms
- print the initial world
- for each step of the simulation
  - update next based on current
  - print the next world
  - switch the roles of current and next

### Counting Neighbors

- Life and death of life forms depends upon the number of various life forms in adjacent cells.

Doing this counting is the same for all life forms so we will implement it in an abstract class, Living.

- **Pseudocode**
  - set the count for all life form types to 0
  - for each of the current cells 8 immediate neighbors
    - if the neighbor is type LifeType
    - then increment the count for LifeType

```java
//Living.java - the superclass for all life forms
abstract class Living {
    abstract Count getCount();
    abstract Living next(World world);
    abstract char toChar(); // character for this form
    void computeNeighbors(World world) {
        world.clearNeighborCounts();
        world.cells[row][column].getCount().set(-1);
        for (int i = -1; i <= 1; i++)
            for (int j = -1; j <= 1; j++)
                world.cells[row+i][column+j].getCount().inc();
    }
    int row, column; //location
}
```

```java
//Fox extends Living
class Fox extends Living {
    Fox(int r, int c, int a) {
        row = r; column = c; age = a;
    }
    Living next(World world) {
        computeNeighbors(world);
        if (fox.neighborCount.get() > 5) { //too many Foxes
            return new Empty(row, column);
        } else if (age > LIFE_EXPECTANCY) { //Fox is too old
            return new Empty(row, column);
        } else if (rabbit.neighborCount.get() == 0) { //starved
            return new Empty(row, column);
        } else {
            return new Fox(row, column, age + 1);
        }
    }
}
```

```java
public String toChar() { return 'F'; } //character for this form
Count getCounter() { return neighborCount; }
private int age;
private final int LIFE_EXPECTANCY = 5;
```
Why must getCount() and neighborCount be repeated in each subclass of Living? Why not just move these definitions to Living?

Answer: There is no way to write a method in Living, that accesses a static field in the subclasses of Living. We need a neighborCount for each of Fox, Rabbit, Grass, and Empty.

```
class Rabbit extends Living {
    Rabbit(int r, int c, int a) {
        row = r;
        column = c;
        age = a;
    }
    ...
    Count getCount() { return neighborCount; }
}
```

```
static Count neighborCount = new Count();
private int age;
private final int LIFE_EXPECTANCY = 3;
```

Why must getCount() and neighborCount be repeated in each subclass of Living? Why not just move these definitions to Living?

Answer: There is no way to write a method in Living, that accesses a static field in the subclasses of Living. We need a neighborCount for each of Fox, Rabbit, Grass, and Empty.

Interfaces

- An interface in Java, is essentially a pure abstract class - all methods are abstract.
- An interface can contain only abstract methods, and constants. There are no instance variables.

```
interface Living {
    Living next(World world);
    char toChar();
    void computeNeighbors(World world);
    Count getCount();
}
```

```
class Fox implements Living {
    // omitted methods/instance variables that are the same as the earlier Fox class.
    public void computeNeighbors(World world) {
        // each class must now define this method
    }
    // an interface doesn’t contain data fields
    // so row and column must be declared here
    // we can now make these members private
    private int row, column;
}
```

Multiple Inheritance

- What happens if a student is also a staff member?

```
Person
  
  Student

  Faculty

  Staff
```

```
Person

  Faculty

  Staff
```

Multiple Inheritance

- What happens if a student is also a staff member?

```
Person

  Faculty

  Staff
```

```
Person

  Faculty

  Staff
```
implements vs extends

• Java does not allow you to extend multiple classes. This avoids some problems related to multiple inheritance (e.g. what happens if you inherit a method from two different parent classes?).
• Java does allow you to implement multiple interfaces.
• Interfaces are about subtyping and polymorphism, whereas, inheriting methods is about code reuse.

interface Person {
    void setAge(int age);
    void setGender(char gender);
    void setName(String name);
    int getAge();
    char getGender();
    String getName();
}

import java.util.Date;

interface Student extends Person {
    void setCollege(String college);
    void setGpa(double gpa);
    void setYear(byte year);
    String getCollege();
    double getGpa();
    byte getYear();
    static final byte FROSH = 1;
    static final byte SOPH = 2;
    static final byte JUNIOR = 3;
    static final byte SENIOR = 4;
}

import java.util.Date;

interface Staff extends Person {
    void setSalary(double salary);
    void setStartDate(Date start);
    void setEndDate(Date end);
    void setSSN(String ssn);
    double getSalary();
    Date getStartDate();
    Date getEndDate();
    String getSSN();
}

Can’t instantiate an interface

• Just as with abstract classes, you can declare variables and parameters to be of an interface type, but you cannot create objects of an interface type.
• Using the declarations on the previous slides, new Staff() would be illegal.
• You could declare a variable or parameter to be of type Staff. This variable would always either be null or refer to an instance of some class that implements the interface Staff.
instanceof

if (x instanceof Shape)
    // x is a Shape, act accordingly
else
    // x is not a Shape, act accordingly
...
Person person = new Student();
...
if (person instanceof Student) {
    Student student = (Student)person;
    // operate on student
}

Illegal Cast

• Some illegal casts can be determined at compile time, others can only be detected at runtime.
• Trying to cast a Person into a String is a compile time error. The compiler knows that String is not a subclass of Person and hence the cast is always illegal.
• Trying to cast a Person into a Student, when the Person is not a Student, is a runtime error - even if YOU can see that this person couldn’t possibly be a Student.

ClassCastException

• The following can only be detected at runtime, even though you can see it is clearly illegal.

Person person = new Person();
Student student = (Student)person;

// GenericArray.java -
//     demonstrate generic array container

class GenericArray {
    public static void main(String[] args) {
        Object[] array = new Object[4];
        array[0] = "String 1";
        array[1] = new Integer(1);
        array[2] = "String 2";
        array[3] = new Integer(2);
        for (int i = 0; i < array.length; i++) {
            if (array[i] instanceof String) {
                String temp = (String)array[i];
                System.out.println("Processing string "+ temp);
                // do something appropriate for the strings
            } else if (array[i] instanceof Integer) {
                Integer temp = (Integer)array[i];
                System.out.println("Processing Integer "+ temp);
                // do something appropriate for an Integer
            } else {
                System.out.println("Unexpected type "+ array[i]);
                // do something to handle unexpected cases
            }
        }
    }
}