Lecture 37: Cloning

Last time:

1. Object
2. Polymorphism and abstract methods
3. Upcasting / downcasting

Today:

1. Safe downcasting
2. `equals` reconsidered
3. Copying and cloning
4. Composition

Safe Downcasting

- Illegal downcasting results in a thrown `ClassCastException` at run-time
- Q: Can we check for the legality of a cast before trying it?
- A: Yes, using `instanceof`
- Example
  - Given: `ArrayList` of university people
  - Want: Print the GPAs of the students
  - Solution approach
    - Iterate through list
    - Print GPAs only of Student
  - Recall the following `Iterator` methods:
    - `hasNext()`
    - `next()`

Example

```
public void printGPAs  (ArrayList<Person> a) {
   Iterator<Person> i = a.iterator();
   Person p;
   while (i.hasNext()) {
       p = i.next();
       if (p instanceof Student)
         System.out.println (((Student) p).getGPA());
   }
}
```

Is p a Student?

If so, cast p to Student (necessary to get access to getGPA method) and print GPA

Object and Upcasting/Downcasting (Recap)

- All objects are derived (directly or indirectly) from Object
- Late binding and inheritance allows you to create polymorphic variables
- When a method in a base class is not provided, the method and class are said to be abstract. Abstract methods may be implemented in (concrete) derived classes
- Run-time information about class / type information of objects can be obtained using **`getClass()`** method and **`instanceof`** operator
- Upcasting of object to superclass type is always safe and done automatically
- Downcasting may not be safe, depending on actual class object belongs to at run-time. Safe downcasting can be done using `instanceof`

Abstract Classes vs. Interfaces

- Interfaces seem like “pure abstract classes”
  - Interfaces contain method prototypes
  - These are similar to abstract methods
- Abstract classes permit some methods to be defined and shared

  Advantage: abstract classes
- A given class can match multiple interfaces, but can only inherit from one class

  Advantage: interfaces
- Which to use?
  - Code to share: use abstract classes
  - Otherwise: interfaces

University Person Recap

```
class: Person
instance variables:
  String name
  String idNum
methods:
  Person( … ) [various]
  String getName( )
  String getIdNum( )
  void setName( String )
  void setIdNum( String )
  String toString( )
  boolean equals( Person )
```

extends Person

```
class: Student
instance variables:
  int admitYear
  double gpa
methods:
  Student( … ) [various]
  int getAdmitYear( )
  double getGpa( )
  void setAdmitYear( int )
  void setGpa( double )
  String toString( )
  boolean equals( Student )
```

extends Person

```
class: Faculty
instance variables:
  int hireYear
methods:
  Faculty( … ) [various]
  int hireYear( )
  void setHireYear( int )
  String toString( )
  boolean equals( Student )
```

`equals()` Reconsidered

- Recall definition of `equals()`
  - … in Person

```
public boolean equals (Person p) {
     return name.equals(p.getName()) &&
            idNum.equals(p.getIdNum());
   }
```

  - … in Student

```
public boolean equals( Student s ) {
     return super.equals(s) &&
            admitYear == s.admitYear &&
            gpa == s.gpa;
    }
```

- What does following do?

```
public static void main (String[] args) {
   Student bob = new Student ("R. Goode", "234-56-7890", 1998, 3.89);
   Faculty bob2 = new Faculty ("R. Goode", "234-56-7890", 2005);
   System.out.println (bob.equals (bob2));
}
```

- `true` is printed!

Why?

- `bob` "is-a" `Student`
- `bob2` "is-a" `Faculty`
- Both "are-a" `Person`
  - `bob2` cannot be cast to `Student`
  - `bob2` can be cast to `Person`
  - So the `Person` version of `equals()` in `bob` is called (overloading!)
- But how can objects in two different classes be equal?

A Better `equals()`

- Take `Object` as input
- Check for non-null-ness of input
- Check that class is correct
- Then do other checks
- For example in `Person`:

```
public boolean equals (Object o) {
    if (o == null)
        return false;
    else if (o.getClass() != getClass())
        return false;
    else {
        Person p = (Person)o;
        return name.equals(p.getName()) &&
            idNum.equals(p.getIdNum());
    }
}
```

- Similar improvements can be made to `Student`, `Faculty`
- Now `bob.equals(bob2)` returns `false`

Copy Constructors Reconsidered

- Copy constructors used to make copies of objects
- Recall copy constructor for Person

```
public Person( Person p ) {
  name = p.name;
  idNum = p.idNum;
}
```

- Does this always do what we want?

Deep Copying of a `Person` Array

```
public static Person[] deepCopy (Person[] a) {
   Person[] r = new Person[a.length];
   for (int i=0; i < a.length; i++)
           r[i] = new Person(a[i]);
   return r;
}
```

- What happens if `a` contains `Student`, `Faculty` objects?
- They are converted into **`Person`** objects; extra info lost
  - Consider:
    - `a[0] = new Student (“BG”,“123-45-6789”,2005,3.2);`
    - `a[1] = new Faculty (“FS”,“111-11-1111”,2003);`
  - `deepCopy(a)` returns `r`:
    - `r[0]` is `Person` object with `name==“BG”, idNum == “123-45-6789”`
    - `r[1]` is `Person` object with `name==“FS”, idNum == “111-11-1111”`
- Why?
  - Person copy constructor creates object in class Person
  - Often, in copying, we want to preserve original class of copied object
  - How to do this

A Better Way to Do Copying

- In addition to copy constructors in classes …
- … include copying capability in objects
  - To make a copy of an object, call object's copy method
  - This way, new copy will be in same class as original object (late binding)
- This object-based approach to copying is called cloning
- `Object` class includes clone method that can be overridden

Defining clone in Person

```
public Person clone () {
    return new Person(this);
}
```

- `this` used to refer to current object
- Copy constructor called on `this`
- Similar `clone() methods` definable for `Student, Faculty`
- New `deepCopy()` method:

```
public static Person[] deepCopy (Person[] a) {
        Person[] r = new Person[a.length];
        for (int i=0; i < a.length; i++)
            r[i] = a[i].clone();
        return r;
}
```

`equals()` and `clone()`: A Summary

- When class inheritance, you may not know the actual classes of objects that you are manipulating. This complicates your job as a programmer.
- Set up your method parameters to be of the most general type that is applicable to your method (e.g. `Object` or `Person` rather than `Student` or `Faculty`).
- Whenever dealing with references to objects in the class hierarchy, use method overriding to produce the proper behavior. (E.g. using `clone()` rather than a copy constructor).

Inheritance vs. Composition

- Inheritance: a way to build new classes out of old ones
  - Objects in subclass inherit data, methods from superclass
  - Object in a subclass “is-a”(n) object in superclass
- Composition: another way to build new classes out of of old
  - Class definitions may include instance variables involving objects from other classes
  - Object in a new class “has-a”(n) object in the original class

Example: Shapes

```
public abstract class Shape {          // Abstract class
   private int color;                  // Color of shape
   …
}

public class Circle extends Shape {  // Concrete class
   private double radius;    // Radius of circle
   private Point center;     // Center of circle
   …
}

public class Rectangle extends Shape {        // Concrete class
   private Point upperLeft; // Upper left corner
   private Point lowerRight;          // Lower right corner
   …
}
```

- Each `Circle`, `Rectangle` "is-a" `Shape` (inheritance)
- Each `Circle` "has-a" `Point` (`center`) (composition)
- Each `Rectangle` "has-a" two `Point`s (`upperLeft`, `lowerRight`)

When To Use Inheritance, Composition?

- Ask “is-a” / “has-a” question
- Answer = “is-a”: inheritance
- Answer = “has-a”: composition