Lecture 28: Inheritance

Last time:
1. Collections: Array List
2. For each looping construct
3. String Buffer

Today:
1. Inheritance

Inheritance

A crucial feature of object-oriented programming languages
- One class (derived class, subclass) is constructed...
- ... by including (extending, inheriting) information...
- ... from another (base class, superclass, parent class)...
- ... and adding new information / redefining existing

Example
- Base class: Clock
  - setTime
  - getTime
  - tick
- Derived class: Alarm Clock
  - Same methods as Clock plus a few additional ones: setAlarm, ring

Can We Avoid Code Copying and therefore redundancy?

- Alarm Clock “IS-A” Clock
- Operations on Clock (e.g. setTime) should be inherited by Alarm Clock
- Alarm Clock should only have to add information specific to alarm clocks
  - setAlarm
  - ring
- Inheritance provides just this capability
Inheritance

- One class (derived class, subclass, child class) is constructed by including (extending, inheriting) information from another (base class, superclass, parent class) then also adding new information and/or redefining existing information.

- To derive a class D from a base class B, use:
  
  ```java
  public class D extends B { … }
  ```

- Example (we will look at this in next two slides):
  
  ```java
  public class Shape
  public class Circle extends Shape
  ```

- Derived class inherits all instance variables, methods from base class. It can also define new instance variables, methods.

- Polymorphism: object in derived class can be used anywhere base class is expected (an `alarmClock` is a `Clock`!)

Inheritance More Generally

- Classes / objects have a natural “is-a” hierarchy
- Object-oriented programming provides mechanisms for exploiting this for
  
  - Code re-use
  - Common operations implemented in super classes
  - Polymorphism
    - Objects in subclasses can be used wherever superclass objects are needed

Example: People at University

- Base class: person
- Derived classes: student, faculty, administrator
- Derived from those: undergrad, grad, instructor, professor,…
University Person Example

Class: Person
Instance variables:
- name: String
- idNum: String

Methods:
- Person(...)
- getName(): String
- getIdNum(): String
- setName(String)
- setIdNum(String)
- toString(): String
- equals(Person)

Class: Student
Instance variables:
- admitYear: int
- gpa: double

Methods:
- Student(...)
- getAdmitYear(): int
- getGpa(): double
- setAdmitYear(int)
- setGpa(double)
- toString(): String
- equals(Student)

Class: Faculty
Instance variables:
- hireYear: int

Methods:
- Faculty(...)
- getHireYear(): int
- setHireYear(int)
- toString(): String
- equals(Faculty)

Memory Layout and Initialization Order

- When you create a new derived class object:
  - Java allocates space for base class instance variables and derived class variables
  - Java initializes base class variables first, and then the derived class variables
- Example:
  Person ted = new Person("Ted Goodman", "111-22-3333");
  Student carole = new Student("Carole Good", "123-45-6789", 2004, 4.0);

Method Overriding

- A derived class can define new instance variables and methods (e.g. hireYear and getHireYear)
- A derived class can also redefine (override) existing methods
  
  ```java
  public class Person {
    ...
    public String toString() { ...
  }
  }
  
  public class Student extends Person {
    ...
    public String toString() { ...
    }
    public String getHireYear() {
      return super.hireYear;
    }
  }
  ```
  
  Student bob = new Student("Bob Good student", "123-45-6789", 2004, 4.0);
  System.out.println("Bob's info: " + bob.toString());
Overriding vs. Overloading

- Overriding: a derived class defines a method with the same name, parameters as base class
- Overloading: two or more methods have the same name, but different parameters

Example:
```java
public class Person {
    public voidsetName(String n) {name = n;}
}
```

Overriding:
```java
public class Faculty extends Person {
    public void setName(String n) {super.setName(n);}
}
```

Overloading:
```java
public class Faculty extends Person {
    public void setName(String first, String last) {
        super.setName(first + " " + last);
    }
}
```

Calling an overridden function

- Possible but use sparingly:
  - Overriding hides methods of the base class (can still access them using `super.methodName()` in subclass, but not in "outside world")
  ```java
  public class Person {
      public String toString() {return "Person";}
  }
  public class Administrator extends Person {
      public String toString() {return "Administrator";}
  }
  ```
  ```java
  public class Faculty extends Person {
      public Student() {
          admitYear = 1988;
      }
      public Student(Student s) {
          super(s);
          admitYear = yr;
      }
      public String toString() {
          return super.toString() + ", admitYear = " + admitYear;
      }
  }
  ```
  ```java
  public class Student( ) {
      private int admitYear;
      // …other methods in part 2
  }
  public Student( Student s ) {
      super(s);
      admitYear = yr;
  }
  public String toString(){/*different def here*/
      return super.toString();  // will use Administrator's def of toString
  }
  public String toString() {return "Student";}
  ```
  ```java
  public Student( ) {
      admitYear = 1988;
  }
  public Student(Student s) {
      super(s);
      admitYear = yr;
  }
  public String toString(){/*different def here*/
      return super.toString();  // will use Administrator's def of toString
  }
  public String toString() {return "Student";}
  ```
  ```java
  public Student( ) {
      admitYear = 1988;
  }
  public Student(Student s) {
      super(s);
      admitYear = yr;
  }
  public String toString(){/*different def here*/
      return super.toString();  // will use Administrator's def of toString
  }
  public String toString() {return "Student";}
  ```
  ```java
  public Student( ) {
      admitYear = 1988;
  }
  public Student(Student s) {
      super(s);
      admitYear = yr;
  }
  public String toString(){/*different def here*/
      return super.toString();  // will use Administrator's def of toString
  }
  public String toString() {return "Student";}
  ```

- Often better to pick a different name rather than overload if you want both.

Derived class: Student

- This calls the default constructor (for base class (Person), Person, to set name and idNum.
- Calls Person constructor.
- Adds additional instance variables.
- copy constructor.
- Calls Person copy constructor.
- Calls Person constructor.
Understanding the Student

- `extends` specifies that `Student` is subclass of `Person`:
  ```java
  public class Student extends Person
  ```
- `super()`:
  - When creating a new Student object, we need to initialize its base class instance variables (from `Person`)
  - This is done by calling `super(...)`. E.g., `super(name, id)` invokes constructor `Person(name, id)`
  - `super(...)` must be the first statement of your constructor
  - If you do not call `super()`, Java will automatically invoke the base class's default constructor
  - If the base class's default constructor is undefined? Error
  - You must use `super(...), new Person(...)`

Shadowing

- Can we override instance variables just like methods?
- Yes, but be careful!
  - Overriding instance variable is called shadowing
  - Shadowing hides instance variables of base class (can still access them using `super.varName` in subclass, but not in "outside world")
  ```java
  public class Person {
    String name;
    ...
  }
  ```
  ```java
  public class Administrator extends Person {
    String name; // name refers to Administrator's name
  }
  ```
  - Confusing! Better to pick a new variable name

Super vs. this

- `super`: refers to the base class
  - Can invoke any base class constructor using `super(...)`
  - Can access data and methods in base class (`Person`) via `super`
  ```java
  E.g., toString(), equals() | invoke the corresponding methods from Person base class using super.toString() and super.equals()
  ```
- `this`: refers to current class / object
  - Can refer to own data and methods using this (usually unnecessary)
  - Can invoke any of its own constructors using `this(...)`. Like `super:
    - Can only be done within a constructor
    - Must be the first statement of the constructor
  Example:
  ```java
  public Faculty(Faculty f) {
    this(f.getName(), f.getIdNum(), f.hireYear);
  }
  ```
  ```java
  public Faculty(String name, int id, int year) {
    super(name, id, year);
  }
  ```
Inheritance and private

- Student inherits all private data (name and idNum) from Person.
- However, private members of base class cannot be accessed directly.

```java
public class Student extends Person {
    public void someMethod() {
        name = "Mr. Foobar";  // Illegal!
    }
    public void someMethod2() {
        setName("Mr. Foobar");  // Ok
    }
}
```

- Why?
  - Although Student inherits from Person...
  - ... they are different classes.

Early vs. Late Binding

- Consider:
  
  ```java
  Faculty carol = new Faculty("Carol Tuffteacher", "999-99-9999", 1995);
  Person p = carol;
  System.out.println( p.toString() );
  ```

- Which version of toString – Person or Faculty – is called?
  - Early (static) binding
    - p is declared to be of type Person.
    - Therefore, the Person version of toString is used.
  - Late (dynamic) binding
    - The object to which p refers was created as Faculty object.
    - Therefore, the Faculty version of toString is used.

- Java uses late binding (C++ by default uses early binding)
  - Early binding is more efficient (decisions about method versions can be made at compile time)
  - Late binding respects encapsulation (object defines its operations when it is created)

Polymorphism

- Java’s late binding makes it possible for a single reference variable to refer to objects of many different types. Such a variable is said to be polymorphic (meaning having many forms).
- Example: Create an array of various university people and print.

```java
Person[] list = new Person[3];
list[0] = new Person( "Col. Mustard", "000-00-0000" );
list[1] = new Student ( "Ms. Scarlet", "111-11-1111", 1998, 3.2 );
list[2] = new Faculty ( "Prof. Plum", "222-22-2222", 1981 );
for ( int i = 0; i < list.length; i++ )
    System.out.println( list[i].toString() );
```
Object

- Recall: inheritance induces "is-a" hierarchy on classes
  - Undergrad "is-a" Student
  - Student "is-a" Person
  - etc.
  - Person "is-a"...?
  - Person "is-a"(n) Object
  - Student "is-a"(n) Object

More on Object

- Special class at top of class inheritance hierarchy
- Defined in java.lang (so available in every program)
- Every class is derived (either directly or indirectly) from Object
  - If a class is not derived from anything, it is automatically derived from Object
    - e.g. public class Foo {...}
      is equivalent to public class Foo extends Object {...}
- Structure of Object
  - No instance variables
  - A number of methods, including:
    - toString()
    - equals (Object o)
      Note: parameter to equals has type Object, so any object can be an argument
  - These methods can (and usually should) be overridden

Class vs. Type Information

- In Java
  - Every object is in one class (the one it was created from using new)
  - Objects may have many types
    - Interfaces
    - Superclasses
  - E.g. consider
    Student bob = new Student();
    Person p = bob;
  - Class of object pointed to by bob, p is Student
  - Type of object can be Student, Person, Object, etc.
**Accessing Class and Type Information**

- Objects can access their class info at run-time
  - `getClass()` method defined in `Object`
  - Returns representation of object's class
  - E.g.: `Person bob = new Person(...); Person ted = new Student(...); if (bob.getClass() == ted.getClass()) // false (ted is really a student)`

- `instanceof` (Java boolean operator, not a method)
  - Returns true if given object "is-a" object of given (class) type
  - E.g.: `Student carol = new Student(...); if (carol instanceof Person) // true, because carol "is-a" Person`

---

**Object Casting**

- Recall casting in primitive types
  - Casting: conversion of elements from one type to another
    - Widening Conversion
      - Every element in source type is a element in destination type
      - Can be done automatically
      - E.g.: `double x = 3.0;` // 3 (int) widening conversion to double
    - Narrowing Conversion
      - Elements in source type are not necessarily elements in the destination type
      - Must use explicit type conversions to perform this casting
      - E.g.: `int x = (int)3.0;` // 3.0 explicitly cast to int
  - Similar notions can be found with object types also
    - Upcasting
      - Casting a reference to a superclass (casting up the inheritance tree)
      - Always done automatically and is always safe
      - Just ignore the parts that were added by the subclass
    - Downcasting
      - Casting a reference to a derived class
      - Requires explicit casting operator, which checks type info at run-time
      - Can cause runtime error

---

**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 Students
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!

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
- New bob.equals(bob2) Returns False

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
- Association: another way to build new classes out of old
  - Class definitions may include instance variables which are objects of other class types
  - Object in a new class “has-a”(n) object in the original class
- Composition: the strongest form of association – when the lifetime of the enclosed object is completely dependant on the lifetime of the object that contains it
Example

```java
public class Base {
    protected void m(int x) { ... }
}

public class Derived extends Base {
    public void m(int x) { ... }
    public int m(int x) { ... }
    public void m(double d) { ... }
}
```

```java
// The following appears in the same package as above
Base b = new Base();
Base d = new Derived();
Derived e = new Derived();
b.m(5);
d.m(6);
d.m(7.0);
e.m(8.0);
```

Error!
- duplicate method declaration
- Overriding: with increased visibility
- Overloading

Error!
- Since d is declared Base, the compiler looks for Base:m(double)
- Doesn't exist! So this does not make it past the compiler, even though Derived:m(double) is defined!

Is Object Abstract?

- No!
- You can create instance of Object using
  ```java
  new Object();
  ```
- But there's not a lot you can do with them

Multiple Inheritance –
and the diamond of death

- Objects of type D now have two copies of whatever was defined in A
- Java uses interfaces to implement what C++ does in multiple inheritance