CMSC 131: Chapter 25 (Supplement)
Iterators and Inheritance

Iterators

ArrayList is inherited from an abstract class called AbstractList.

- Java provides many different data structures that are inherited from AbstractList, e.g. linked lists, binary trees, hash tables.
- Most of these structures do not provide a method like get(i) for accessing elements.
- They all provide a device for enumerating all the elements of the data structure: Iterator. An iterator is an object that allows you to enumerate the elements of a list, one by one.

How Iterators work: Let list be an ArrayList (or any class inherited from AbstractList).

Iterator x = list.iterator(): Creates a new iterator object for list. It is positioned at the start of the list.

x.next(): Returns the next element of the list (an Object), and advances the iterator. (Throws an exception if none left.)

x.hasNext(): Returns true if more elements remain in the list.

Example

Print all the elements of an ArrayList:

    public static void print( ArrayList a ) {
        Iterator it = a.iterator();
        while ( it.hasNext() )
            System.out.println( it.next() )
    }

Perform a linear search in an ArrayList:

    public static boolean find2( ArrayList a, Object q ) {
        Iterator it = a.iterator();
        while ( it.hasNext() )
            if ( it.next().equals( q ) ) return true;
        return false;
    }
Inheritance

Inheritance: is the process by which one new class, called the derived class, is created from another class, called the base class.

- The derived class is also called: subclass or child class.
- The base class is also called: superclass or parent class.

Motivation: In real life objects have a hierarchical structure: (figure omitted)

We want to do the same with our program objects.

Inheritance

Object Inheritance: What does inheritance mean within the context of object-oriented programming?

Suppose a derived class, Circle, comes from a base class, Shape:

- Circle should have all the instance variables that Shape has.
  (E.g., Shape stores a color, and thus, Circle stores a color.)
- Circle should have all the methods that Shape has (E.g., Shape has an accessor, getColor(), and thus, Circle has getColor()).
- Circle is allowed to define new instance variables and new methods that are particular to it:
  (New) Circle Instance variables: Center, radius.
  (New) Methods: draw(), getArea(), getPerimeter().

Code reuse: Code/Data that is common to all the derived classes can be stored in the base class. This allows us to avoid code duplication, and so makes development and maintenance easier.
Example: University People

Consider an example for a university database, which stores information on various people at the university. The various objects form a hierarchy: (figure omitted)

We will consider the design of the Person, Student, and Faculty classes.

These classes will be very simple (almost trivial). Watch for the relationships between these classes.

Base Class: Person (Part 1)

package university;

public class Person {
    private String name; // person's name
    private String idNum; // ID number

    public Person() {
        name = "No Name";
        idNum = "000-00-0000";
    }

    public Person( String n, String id ) {
        name = n;
        idNum = id;
    }

    public Person( Person p ) {
        name = p.name;
        idNum = p.idNum;
    }
    // ...other methods in part 2
}

Base Class: Person (Part 2)

public class Person {
    private String name; // person's name
    private String idNum; // ID number

    // ...constructors in part 1

    public String getName() { return name; }
    public String getIdNum() { return idNum; }
    public void setName( String n ) { name = n; }
    public void setIdNum( String id ) { idNum = id; }
    public String toString() {
        return "[" + name + "]" + idNum;
    }

    public boolean equals( Person p ) {
        return name.equals( p.name ) && idNum.equals( p.idNum );
    }
}
Derived Classes: Student and Faculty

We derive two classes Student and Faculty. Each class inherits all the data and methods from Person, and adds data and methods that are particular to its particular function. (figure omitted)

**Student**: In addition to name and ID, has admission year and GPA.
**Faculty**: In addition to name and ID, has the year they were hired.

Derived Class Structure

**Person**: (base class)

- **Instance Data**: Name and ID-number.
  - String name
  - String idNum

- **Methods**:
  - Constructors: default, standard, copy constructors.
  - Accessors/Setters: getName(), setName(), getIdNum(), setIdNum().
  - Standard methods: toString(), equals().

**Student**: (derived from Person)

- **Instance Data**: Admission year and GPA.
  - int admitYear
  - double gpa

- **Methods**: (same structure as Person)

**Faculty**: (derived from Person)

- **Instance Data**: Year hired.
  - int hireYear

- **Methods**: (same structure as Person)
Derived class: Student (Part 1)

```java
package university;
public class Student extends Person {
    private int admitYear;
    private double gpa;

    public Student() {
        super();
        admitYear = -1;
        gpa = 0.0;
    }
    public Student(String n, String id, int yr, double g) {
        super(n, id);
        admitYear = yr;
        gpa = g;
    }
    public Student(Student s) {
        super(s);
        admitYear = s.admitYear;
        gpa = s.gpa;
    }
    // ...other methods in part 2
}
```

Dissecting the Student Class

**Extends:** To specify that `Student` is a derived class (subclass) of `Person` we add the descriptor "extends" to the class definition:

```java
public class Student extends Person { ... }
```

**super( ):** When initializing a new `Student` object, we need to initialize its base class (or superclass). This is done by calling `super( ... )`. For example, `super( name, id )` invokes the 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.
- What if the base class's default constructor is undefined? Error.
- You must use "super( ... )", not "Person( ... )".
Memory Layout and Initialization Order

When you create a new derived class object:

- Java allocates space for both the base class instance variables and the derived class variables.
- Java initializes the base class variables first, and then initializes the derived class variables.

Example:

```java
Person ted = new Person("Ted Goodman", "111-22-3333");
Student bob = new Student("Bob Goodstudent", "123-45-6789", 2004, 4.0);
```

(figure omitted)

Derived class: Student (Part 2)

```java
public class Student extends Person {
    private int admitYear;
    private double gpa;

    // ... constructors in part 1 ...

    public int getAdmitYear() { return admitYear; }
    public double getGpa() { return gpa; }

    public void setAdmitYear(int yr) { admitYear = yr; }
    public void setGpa(double g) { gpa = g; }

    public String toString() {
        return super.toString() + " " + admitYear + " " + gpa;
    }

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

Inheritance: Since Student is derived from Person, a Student object can invoke any of the Person methods, it inherits them.

```java
Student bob = new Student( "Bob Goodstudent", "123-45-6789", 2004, 4.0 );
System.out.println( "Bob's name is " + bob.getName() );
bob.setName( "Robert Goodstudent" );
System.out.println( "Bob's new info: " + bob.toString() );
```

A Student "is a" Person:

- By inheritance a Student object is also a Person object. We can use a Student reference anywhere that a Person reference is needed.
  ```java
  Person robert = bob;  // Okay: A Student is a Person
  ```
- However, we cannot do it the other way around. (A Person need not be a Student.)
  ```java
  Student bob2 = robert;  // Error: Cannot convert Person to Student
  ```

Recap:

- Inheritance is when one class (derived class or subclass) is defined from another class (the base class or superclass).
- To derive a class D from a base class B, we use the declaration:
  ```java
  public class D extends B { ...
  ```
- The derived class inherits all the instance variables and the methods from the base class. It can also define new instance variables and new methods.
- When a derived class is initialized, it can use super( ... ) to invoke the constructor for the base class.
- A derived class can explicitly refer to entities from the base class using super. For example, super.toString( ) invokes the base class's toString method.
- A reference to a derived class can be used anywhere where a reference to the base class is expected.

  **Remember:** A Student "is a" Person.
Derived Class: Faculty

```java
class Faculty extends Person {
    private int hireYear; // year when hired

    public Faculty() { super(); hireYear = -1; }
    public Faculty(String n, String id, int yr) {
        super(n, id);
        hireYear = yr;
    }
    public Faculty(Faculty f) {
        this(f.getName(), f.getIdNum(), f.hireYear);
    }
    int getHireYear() { return hireYear; }
    void setHireYear(int yr) { hireYear = yr; }

    public String toString() {
        return super.toString() + " " + hireYear;
    }
    public boolean equals(Faculty f) {
        return super.equals(f) && hireYear == f.hireYear;
    }
}
```

Overriding Methods

**New Methods:** A derived class can define entirely new instance variables and new methods (e.g. hireYear and getHireYear()).

**Overriding:** A derived class can also redefine existing methods.

```java
public class Person {
    ...
    public String toString() { ... }
}
public class Student extends Person {
    ...
    public String toString() { ... }
}
```

```java
Student bob = new Student("Bob Goodstudent", "123-45-6789", 2004, 4.0);
System.out.println("Bob's info: " + bob.toString());
```
Overriding and Overloading

Don't confuse method overiding with method overloading.

*Overriding:* occurs when a derived class defines a method with the *same name* and *parameters* as the base class.

*Overloading:* occurs when two or more methods have the *same name*, but have *different parameters* (different signature).

**Example:**
```java
public class Person {
    public void setName(String n) { name = n; }
    ...
}
public class Faculty extends Person {
    public void setName(String n) {
        super.setName("The Evil Professor " + n);
    }
    public void setName(String first, String last) {
        super.setName(first + " " + last);
    }
}
```

**Overriding Variables: Shadowing**

We can override methods, can we override instance variables too?

**Answer:** Yes, it is possible, but not recommended.

- Overriding an instance variable is called *shadowing*, because it makes the base instance variables of the base class inaccessible. (We can still access it explicitly using `super.varName`).

  ```java
  public class Person {
      String name;
      // ...
  }
  public class Staff extends Person {
      String name;
      // name refers to Staff's name
  }
  ```

- This can be confusing to readers, since they may not have noticed that you redefined `name`. Better to just pick a new variable name.
super and this

super: refers to the base class object.

- We can invoke any base class constructor using super(...).
- We can access data and methods in the base class (Person) through super. E.g., toString() and equals() invoke the corresponding methods from the Person base class, using super.toString() and super.equals().

this: refers to this object.

- We can refer to our own data and methods using “this,” but this usually is not needed.
- We can invoke any of our own constructors using this(...). As with the super constructor, this can only be done within a constructor, and must be the first statement of the constructor.

Example:
public Faculty(Faculty f) {
    this(f.getName(), f.getIdNum(), f.hireYear);
}

Inheritance and Private

Inheritance and private members:

- Student objects inherit all the private data (name and idNum).
- However, private members of the base class cannot be accessed directly.

Example: (Recall that name is a private member of Person.)
public class Student extends Person {
    ...
    public void someMethod() { name = "Mr. Foobar"; } // Illegal!
    public void someMethod2() { setName("Mr. Foobar"); } // Okay
}

Why is this? After you have gone to all the work of setting up privacy, it wouldn't be fair to allow someone to simply extend your class and now have direct access to all the private information.
Protected and Package Access

The derived class cannot access private base elements. So can a base class grant any special access to its derived classes?

Special Access for Derived Classes:

**Protected**: When a class element (instance variable or method) is declared to be protected (rather than public or private) it is accessible:

- to any derived class (and hence to all descendants), and
- to any class in the same package.

Example:
```java
protected void someMethod() { ... } // has protected access
```

**Package**: When a class element not given any access modifier (private, public, protected) it is said to have package access. It is accessible:

- to any class in the same package.

Example:
```java
void someOtherMethod() { ... } // has package access
```

Access to Base Class Elements

**Which should I use?**: private, protected, package, or public

**Public**:
- Methods of the object's public interface.
- Constant instance variables (static final).

**Private**:
- Instance variables (other than constants).
- Internal helper/utility methods (not intended for use except in this class).

**Protected/Package**:
- Internal helper/utility methods (for use in this class and related classes).

**Note**: Some style gurus discourage the use of protected. Package is safer, since any resulting trouble can be localized to the current package.
Access Modifiers

```
package fooBar;
public class A {
    public int vPub;
    protected int vProt;
    int vPack;
    private int vPriv;
}

package fooBar;
public class B {
    can access vPub;
    can access vProt;
    can access vPack;
    cannot access vPriv;
}

package fooBar;
public class C extends A {
    can access vPub;
    can access vProt;
    can access vPack;
    cannot access vPriv;
}

package fooBar;
public class D extends A {
    can access vPub;
    can access vProt;
    cannot access vPack;
    cannot access vPriv;
}

package fooBar;
public class E {
    can access vPub;
    cannot access vProt;
    cannot access vPack;
    cannot access vPriv;
}
```