Programming Languages and Techniques (CIS120e)

Lecture 22
Nov. 1, 2010

Subtyping, Generics, and Collections

Announcements

• Midterm 2 is Friday, November 12th

Review: Interfaces

• An interface represents a point of view about an object
• Classes can implement multiple interfaces

Subtyping

1. Interface extension
2. Class extension (Simple inheritance)
3. Object – the root of the type hierarchy
Interface Extension

- Build richer interface hierarchies by extending existing interfaces.

```java
public interface Displaceable {
    double getX();
    double getY();
    void move(double dx, double dy);
}

public interface Area {
    double getArea();
}

public interface Shape extends Displaceable, Area {
    Rectangle getBoundingBox();
}
```

The Shape type includes all the methods of Displaceable and Area, plus the new getBoundingBox method.

```
Note the use of the “extends” keyword.
```

Class Extension

- Classes, like interfaces, can also extend one another.
  - Unlike interfaces, a class can extend only one other class.
- The extending class inherits copies all of the state and methods of its superclass, and may include additional fields or methods.
  - This captures the “is a” relationship between objects (e.g., a Car is a Vehicle).
  - Class extension should never be used when “is a” does not relate the subtype to the supertype.

```java
class D {
    private int x;
    private int y;
    public int addBoth() { return x + y; }
}
class C extends D { // every C is a D
    private int z;
    public int addThree() { return addBoth() + z; }
}
```

Simple Inheritance*

- In this simple inheritance, the subclass only adds new fields or methods.
- This facility can be used to share common code among implementations of related classes.
- Example: Observe that Point, Circle, and Rectangle have identical code for getX(), getY(), and move() methods when implementing Displaceable.

```
*Java has other features related to inheritance (some of which we will discuss later in the course):
  * A subclass might override (re-implement) a method already found in the superclass.
  * A class might be “abstract” – i.e., it does not provide implementations for all of its methods
    -- the subclass must provide them instead
These features are hard to use properly and the need for them arises in special cases (such as when implementing a library or overriding equals and toString). We recommend avoiding all forms of inheritance (even “simple inheritance”) when possible – prefer interfaces and composition (see Lec22/v3/Main.java for an example). But especially avoid overriding.
```

Interface Hierarchy

- Shape is a subtype of both Displaceable and Area.
- Circle and Rectangle are both subtypes of Shape, and, by transitivity, both are also subtypes of Displaceable and Area.
- Note that one interface may extend several others.
  -- Interfaces do not necessarily form a tree, but the hierarchy has no cycles.
- See Lec22/v1/Main.java for an example.
Example of Simple Inheritance

See: Lec22/v2/Main.java

Inheritance: Constructors

- Constructors cannot be inherited (they have the wrong names!)
  - Instead, a subclass invokes the constructor of its super class using the keyword ‘super’.
  - Super must be the first line of the subclass constructor, unless the parent class constructor takes no arguments, in which it is OK to omit the call to super (it is called implicitly).

```java
class D {
    private int x;
    private int y;
    public D (int initX, int initY) { x = initX; y = initY; }
    public int addBoth() { return x + y; }
}
class C extends D {
    private int z;
    public C (int initX, int initY, int initZ) {
        super(initX, initY);
        z = initZ;
    }
    public int addThree() {return (addBoth() + z); }
}
```

Object

- Object is the root of the class tree.
  - Classes that leave off the “extends D” clause implicitly extend Object
  - Arrays also implement the methods of Object
  - This class provides methods useful for all objects to support.
- Object is also the top type of the subtyping hierarchy.

```java
public class Object {
    Object clone () {
        ... // return a copy of this object
    } 
    boolean equals(Object o) {
        ... // test for “structural” equality
    }
    String toString() {
        ... // return a string representation
    }
    // other methods omitted
}
```

Subtype Hierarchy in Detail

- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object if implicit)
- Interface types are “subtypes by fiat” of Object
Subtyping

Object

classes (form a tree)

interfaces

Displaceable

Area

Shape

DisplaceableImpl

Point

Circle

Rectangle

Type C is a subtype of D if D is reachable from C by following zero or more edges upwards in the hierarchy.
- e.g. Circle is a subtype of Area, but Point is not

The Java Collections Library

A case study in subtyping and generics.
(Also very useful!)

Interfaces* of the Collections Library

public interface Collection<E> extends Iterable<E> {
    // basic operations
    int size();
    boolean isEmpty();
    boolean add(E o);
    boolean remove(Object o); // why not E?*
    boolean contains(Object o);
    // bulk operations
    ...
}

• We’ve already seen this interface in the OCaml part of the course.

• Most collections are designed to be mutable (like our doubly-linked lists)

* Why not E? Internally, collections use the equals method to check for equality – membership is determined by a equals, which does not have to be false for objects of different types. Most applications only store and remove one type of element in a collection, in which case this subtlety never becomes an issue.

Reminder: Collection<E> is a generic collection type, in OCaml we’d write: ‘e collection

*The collections library includes several other interfaces too.
Java Packages

- Java code can be organized into packages that provide namespace management.
  - Somewhat like OCaml’s modules
  - Packages contain groups of related classes and interfaces.
  - Packages are organized hierarchically in a way that mimics the file system’s directory structure.
- A .java file can import (parts of) packages that it needs access to:

```java
import org.junit.Test;  // just the JUnit Test class
import java.util.*;   // everything in java.util
```

- Important packages:
  - java.lang, java.io, java.util, java.math, org.junit
- See documentation at: http://download.oracle.com/javase/6/docs/api/index.html

## Reading Java Docs

1. Collection<E>
2. List<E> and Set<E>
3. Iterable<E> and Iterator<E>

## While Loops

Syntax:

```java
// repeat body until condition becomes false
while (condition) {
  body
}
```

Example:

```java
List<Book> shelf = ...  // create a list of Books

// iterate through the elements on the shelf
ListIterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
  Book book = iter.next();
  catalogue.addInfo(book);
  numBooks = numBooks+1;
}
```
For Loops

**syntax:**
```java
for (init-stmt; condition; next-stmt) {
    body
}
```

**equivalent while loop:**
```java
init-stmt;
while (condition) {
    body
    next-stmt;
}
```

**example:**
```java
int[] arr = ... // create an array of ints

// count the non-zero elements of the array
for (int i = 0; i < arr.length; i++) {
    if (arr[i] != 0) cnt = cnt+1;
}
```

For-each Loops

**syntax:**
```java
// repeat body until condition becomes false
for (type var : coll) {
    body
}
```

**example:**
```java
List<Book> shelf = ... // create a list of books

// iterate through the elements on a shelf
for (Book book : shelf) {
    catalogue.addInfo(book);
    numBooks = numBooks+1;
}
```

For-each Loops (Cont’d)

**Another example:**
```java
int[] arr = ... // create an array of ints

// count the non-null elements of an array
for (int elt : arr) {
    if (elt != 0) cnt = cnt+1;
}
```

For-each can be used to iterate over arrays or any class that implements the `Iterable<E>` interface (notably `Collection<E>` and its subinterfaces).

Subtyping and Inheritance

1. Interaction of generics and subtyping
Subtyping and Generics

```java
List<String> ls = new ArrayList<String>();     // OK
List<Object> lo = ls;                         // OK?
lo.add(new Object());
String s = ls.get(0);                        // oops!
```

- **Java generics are invariant:**
  - Subtyping of arguments to generic types does not imply subtyping between the instantiations:

```
Object
   |
   |
   |
String
   |
   |
   |
List<String>
```

*Hardest part to learn about generics and subtyping...*