Interfaces, Abstraction, & Comparisons

Lecture 10
CS211 – Fall 2006

Announcements

- Finding a partner
  - Check the newsgroup
    - Just post a message asking for a partner
  - Use signup sheet
    - We'll put you in contact with potential partners
- Upcoming Prelim I
  - Thu, Oct 12, 7:30-9:00pm
    - Same week as Fall Break
  - If you have a conflict, notify Course Administrator (Kelly Patwell; see website)
- Assignment strategy
  - Don't wait until the last minute to submit files to CMS
    - CMS gets busy
    - We try to be reasonable about a few minutes late
- Lateness policy
  - If any of your assignment is late then the whole assignment is late

Recall

- Upcasting and downcasting
  - Cause *no change* to an object at all
  - Why needed?
    - Provides information to compiler for type checking
    - Provides documentation
- Static type
  - Is the declared type of a variable or expression
- Dynamic type
  - Is the actual type of an object
  - Determined for all time when object is *created*
- Upcasting (moving *up* in the type hierarchy)
  - Is always OK
  - Relation between types is checked at compile time
  - No runtime check needed
- Downcasting (moving *down* in the type hierarchy)
  - Is sometimes OK
  - Relation between types is checked at compile time
  - Actual types are checked at runtime

Upcasting with Interfaces

- Java allows upcasting:

```
IPuzzle p1 = new ArrayPuzzle();
IPuzzle p2 = new IntPuzzle();
```

- Static types of right-hand side expressions are ArrayPuzzle and IntPuzzle
- Static type of left-hand side variables is IPuzzle
- Lhs static type is super type of rhs static types, so this is upcasting and is OK

Code using IPuzzle Interface

Puzzle code

```
interface IPuzzle {
  int tile(int r, int c);
  ...
}
class IntPuzzle implements IPuzzle {
  public int tile(int r, int c) {...}
  ...
}
class ArrayPuzzle implements IPuzzle {
  public int tile(int r, int c) {...}
  ...
}
```

Client code

```
public static void display(IPuzzle p){
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        System.out.println(p.tile(r,c));
}}
```

Method Dispatch

```
public static void display(IPuzzle p){
  for (int r = 0; r < 3; r++)
    for (int c = 0; c < 3; c++)
      System.out.println(p.tile(r,c));
}}
```

- Which tile method is invoked?
  - Depends on dynamic type of object p (IntPuzzle or ArrayPuzzle)
  - We don't know what this dynamic type is, but whatever it is, we know it has a tile method (since any class that implements IPuzzle must have a tile method)
- At compile-time
  - Check that the static type of p (namely IPuzzle) has a tile method with the right *signature*
- At runtime
  - Go to the object that is the value of p, find its dynamic type, look up its tile method
- The compile-time check guarantees that an appropriate tile method exists

Important Note

Upcasting and downcasting do not change the object

- They merely allow it to be viewed at compile time as a different static type

Another Use of Upcasting

- Heterogeneous Data Structures
  - Example:
    ```
    IPuzzle[] pzls = new IPuzzle[9];
    pzls[0] = new IntPuzzle();
    pzls[1] = new ArrayPuzzle();
    ```
  - An expression pzls[i] is of type IPuzzle
  - Objects created on right hand sides are subtypes of IPuzzle

Java instanceof

- Example
  ```
  if (p instanceof IntPuzzle)
    {...}
  ```
  - True if dynamic type of p is a subtype of IntPuzzle
  - Often used to check if a downcast will succeed

- Example
  - Suppose twist is a method implemented only in IntPuzzle

```
void twist(IPuzzle[] pzls) {
  for (int i = 0; i < pzls.length; i++) {
    if (pzls[i] instanceof IntPuzzle) {
      IntPuzzle p = (IntPuzzle)pzls[i];
      p.twist();
}}}
```

Avoid Useless Downcasting

bad

```
void moveAll(IPuzzle[] pzls) {
  for (int i = 0; i < pzls.length; i++) {
    if (pzls[i] instanceof IntPuzzle)
      ((IntPuzzle)pzls[i]).move("N");
    else ((ArrayPuzzle)pzls[i]).move("N");
}}
```

good

```
void moveAll(IPuzzle[] pzls) {
  for (int i = 0; i < pzls.length; i++)
    pzls[i].move("N");
}
```

Subinterfaces

- Suppose you want to extend the interface to include more methods
  - IPuzzle:
    scramble, move, tile
  - ImprovedPuzzle:
    scramble, move, tile, samLoyd

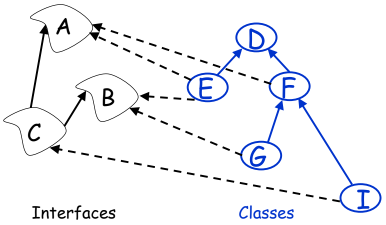
- Two approaches
  - Start from scratch and write an interface
  - Extend the IPuzzle interface

Extending an Interface

```
interface IPuzzle {
   void scramble();
   int tile(int r, int c);
   boolean move(char d);
}
interface ImprovedPuzzle extends IPuzzle {
   void SamLoyd();
}
```

- IPuzzle is a superinterface of ImprovedPuzzle
- ImprovedPuzzle is a subinterface of IPuzzle
- ImprovedPuzzle is a subtype of IPuzzle
- An interface can extend multiple superinterfaces
- A class that implements an interface must implement all methods declared in all superinterfaces

Class Hierarchy is Part of Type Hierarchy



```
interface C extends A,B {...}
class E extends D implements A,B {...}
class F extends D implements A {...}
class G extends F implements B {...}
class I extends F implements C {...}
```

Conclusion

- Interfaces have two main uses
  - Software engineering: good fences make good neighbors
  - Subtyping
- Subtyping is a central idea in programming languages
  - Inheritance and interfaces are two methods for creating subtype relationships in Java

Abstraction

- Abstraction = hiding of unnecessary detail
  - Helpful approach to building complex systems
  - Used all the time in the real world
    - Most people treat a battery as an abstraction
      - It makes electricity—I don't know or care how it works
    - Scientists who work for, say, Duracell have a very different view of batteries
    - Workers on Duracell's assembly-line have yet another abstraction for *battery*

Some Abstractions used in Programming

- Function abstraction
  - A function (method, in Java) provides an abstraction for a piece of code
  - A piece of code becomes much easier to use once it's *packaged* as a function
  - A client can use the function without knowing about details of the code
  - Example: sorting
    - Easy to use Arrays.sort(...) even without knowing how it's coded
- Data abstraction
  - Also called an ADT (Abstract Data Type)
  - Hides information about how data is represented
  - A client can use an ADT without knowing about details of its code
  - Example: set
    - Java provides a Set interface
    - Java provides several classes that implement Set (e.g., TreeSet, HashSet)

Benefits of Abstraction

- Abstraction barrier
  - Client can ignore implementation details
  - Implementer can change/improve the implementation without breaking the client's code
- Client & implementer can work independently
- In case of error, either
  - Client is misusing the abstraction
    - Violating the abstraction barrier (i.e., using knowledge of the implementation)
    - Misusing the abstraction's *interface*
  - Implementer has failed to implement the abstraction correctly

Java Support for Abstraction

- Visibility qualifiers (support for ADT's *abstraction barrier*)
  - public
    - Visible from anywhere
  - private
    - Only visible from current class
  - protected
    - Visible from class and its subclasses
  - <default>
    "package visibility"
    - Visible from any class in same *package*
- Java interfaces (support for clear specification of ADT's *interface*)
  - Says nothing about implementation
  - Specifies a "contract" between client and implementer

Comparison

- Something that we do a lot
- Can compare all kinds of data with respect to all kinds of comparison relations
  - Identity
  - Equality
  - Sorting Order
  - Lots of others

Identity vs. Equality

- For primitive types (e.g., int, long, float, double, boolean)
  - == and != are equality tests
- For reference types (i.e., objects)
  - == and != are identity tests
  - In other words, they test if the references indicate the same address in the Heap
- For equality of objects: use the equals( ) method
  - equals( ) is defined in class Object
  - Any class you create inherits equals( ) from its parent class, but you can override it (and probably want to)

Identity vs. Equality for Strings

- Quiz: What are the results of the following tests?
  - "hello".equals("hello") **true**
  - "hello" == "hello" **true**
  - "hello" == new String("hello") **false**

Order

- For numeric primitives (e.g., int, float, long, double)
  - Use <, >, <=, >=
- For reference types that correspond to primitive types
  - As of Java 5.0, Java does Autoboxing and Auto-Unboxing of Primitive Types
  - This means, for example, that an Integer is automatically converted into an appropriate int whenever necessary (and vice versa)
- For all other reference types
  - <, >, <=, >= do not work
    - Not clear you want them to work: suppose we compare People
      - Compare by name?
      - Compare by height? weight?
      - Compare by SSN? CUID?
  - Java provides Comparable interface
    - Or can use a Comparator

Comparable Interface

```
interface Comparable {
  int compareTo(Object x);
}
```

- (Note: this is Java 1.4 – Java 5.0 has *generics*)
- x.compareTo(y) returns a negative, zero, or positive int based on whether x is less-than, equal-to, or greater-than y, respectively
- less-than, equal-to, and greater-than are *defined* for a class by its implementation of compareTo

Example

- To compare people by weight:

```
class Person implements Comparable {
  private int weight;
  ...
  public int compareTo(Object obj) {
    return this.weight - ((Person)obj).weight;
  }
  public boolean equals(Object obj) {
    return obj instanceof Person &&
      ((Person)obj).weight == weight;
  }
}
```

Consistency

If a class has an equals method and also implements Comparable, then it is advisable (but not enforced) that

a.equals(b)

exactly when

a.compareTo(b) == 0

Odd behavior can result if this is violated

Generic Code

- The Comparable interface allows generic code for sorting, searching, and other operations that only require comparisons

```
static void mergeSort(Comparable[] a) {...}
static void bubbleSort(Comparable[] a) {...}
```

- The sort methods do not need to know what they are sorting, only how to compare elements

Generic Code Example

- Finding the max element of an array

```
//return max element of an array
static Comparable max(Comparable[] a) {
  //throws ArrayIndexOutOfBoundsException
  Comparable max = a[0];
  for (Comparable x : a) {
    if (x.compareTo(max) > 0) max = x;
  }
  return max;
}
```

- What is the max element? Whatever compareTo says it is!

Another Example

- Lexicographic comparison of Comparable arrays
- For int arrays, a < b lexicographically iff either:
  - a[i] == b[i] for i < j and a[j] < b[j]; or
  - a[i] == b[i] for all i < a.length, and b is longer

```
//compare two Comparable arrays lexicographically
static int arrayCompare(Comparable[] a, Comparable[] b) {
  for (int i = 0; i < a.length && i < b.length; i++) {
    int x = a[i].compareTo(b[i]);
    if (x != 0) return x;
  }
  return b.length - a.length;
}
```

Comparable Interface Update

- Java 5.0 allows the use of "*Generic Types*"
  - Better name might be *parameterized types*
  - Here's the Java 5.0 Comparable interface

```
interface Comparable<T> {
  int compareTo(T x);
}
```

  - Note that compareTo is only defined for arguments of type T
    - An attempt to use a different type is caught at *compile time*

Example

- In the Java source code, class String looks sort of like this (other interfaces are also implemented):

```
public final class String implements Comparable<String>{
  public int compareTo (String s) {...}
...}
```

- Code such as "hello".compareTo(new Integer(3)) generates a compile-time error
  - This implies that the runtime code can be more efficient

Using Comparable for Sorting

- Sorting of an array is provided as part of the Java Collections Framework

```
import java.util.Arrays;
...
String[] names;
...
Arrays.sort(names)
```

- This works for arrays of type *comparableType*[ ] (i.e., the base type must implement the Comparable interface)
- (Class java.util.Arrays also contains sort methods for arrays of type *primType*[ ] for each of the primitive types)

Defining a "Natural Ordering"

- An object's natural ordering is determined by its compareTo method
  - For Java to know that an class can be compared, the class must implement the Comparable interface
- Java provides tools to work with objects of type Comparable
  - Examples: sort, binarySearch

```
public class Person
   implements Comparable<Person> {
   private String name;
   private int id, height, weight;

   public Person (String name, int id,
                  int height, int weight) {
      this.name = name; this.id = id;
      this.height = height;
      this.weight = weight;
   }

   public int compareTo (Person p) {
      return id - p.id;
   }
```

"Unnatural" Ordering

- The ordering given by compareTo is considered to be the *natural ordering* for a class
- Sometimes you need to sort based on a different ordering
  - Example: we may normally sort students by CUID, but we might want to produce a list alphabetized by name

```
interface Comparator<T> {
  int compare (T x, T y);
}
```

- Can use a Comparator (a class that implements the Comparator interface)

Arrays.sort(students,comparator)

- String, for example, has a predefined Comparator:

String.CASE_INSENSITIVE_ORDER

Comparators for the Person Class

```
class NameComparator implements Comparator<Person> {
   public int compare (Person p, Person q) {
      return p.getName().compareTo(q.getName());
   }
}

class HeightComparator implements Comparator<Person> {
   public int compare (Person p, Person q) {
      return p.getHeight() - q.getHeight();
   }
}

class WeightComparator implements Comparator<Person> {
   public int compare (Person p, Person q) {
      return p.getWeight() - q.getWeight();
   }
```

Sorting an Array of Persons

- Sort by ID (this is the natural ordering)
- Sort by name
- Sort by height
- Sort by weight

```
import java.util.Arrays;
Person[] p = ...

Arrays.sort(p);

Arrays.sort(p, new NameComparator());

Arrays.sort(p, new HeightComparator());

Arrays.sort(p, new WeightComparator());
```