Interfaces & Java Types

Lecture 9
CS211 – Fall 2006

Announcements

- A3 (due Oct 4) is not yet ready
  - It should be available by early next week (Monday, I hope)
- A2 grades are not yet available
  - Grading session is tonight

Recall

```
class Puzzle {
   //representation of a puzzle state
   private int state;
   //create a new random instance
   public void scramble() {...}
   //say which tile occupies a given position
   public int tile(int r, int c) {...}
   //move a tile
   public boolean move(char c) {...}
}
```

```
class EPuzzle extends Puzzle {
   private int numMoves = 0;
   public void scramble() {...}
   public boolean move(char d) {...}
   public void printNumMoves() {...}
}
```

- Problem:
  - Methods scramble and move in EPuzzle need access to state
- One solution is to change the declaration of state to protected instead of private
  - Methods of EPuzzle can then access state
  - But this is "cheating"

Another Solution

- Suppose subclass S overrides a method m in its superclass
- Methods in subclass S can invoke an *overridden method of superclass* as

  super.m()

- Caveats
  - Cannot compose super many times as in super.super.m()
  - Static binding: super.m is resolved at *compile-time*, so no object look-up at runtime

Another Definition of EPuzzle

```
class EPuzzle extends Puzzle {
   protected int numMoves = 0;
   ...
   public void scramble() {
      super.scramble();
      numMoves = 0;
   }
   public boolean move(char d){
      boolean p = super.move(d);
      if (p) numMoves++; //legal move?
      return p;
   }
}
```

- This version does not need protected access to field state!

Subtypes

- Inheritance gives a mechanism in Java for creating subtypes
  - (Java interfaces are another such mechanism)
- If class B extends class A then B is a subtype of A
- Examples

```
Puzzle p = new EPuzzle();   //up-casting (always legal)
EPuzzle e = (EPuzzle)p;     //down-casting (checked at runtime)
```

Unexpected Consequence

- A method that overrides a superclass method cannot have more restricted access than the superclass method

```
class A {
   public int m() {...}
}

class B extends A {
   private int m() {...}     // Illegal! (see below)
}

A supR = new B();        // Upcasting (always legal)
supR.m(); // Will invoke private method in class B at runtime!
```

Shadowing Variables

- Like overriding, but for *fields* (i.e., variables) instead of *methods*
  - Superclass: variable v of some type
  - Subclass: variable v perhaps of some other type
  - Method in subclass can access shadowed variable by using super.v
- Variable references are resolved using static binding (i.e., at compile-time), not dynamic binding (i.e., not at runtime)
  - Variable reference r.v
    - Uses static type of the variable r, not runtime type of the object referred to by r
  - Note that this behavior is different than it is for methods
- Shadowing variables is bad medicine and should be avoided

Constructors

- No overriding of constructors: each class has its own constructor
- Superclass constructor can be invoked explicitly within subclass constructor by invoking super() with parameters as needed
- Can invoke other constructors of the *same class* using this()
- When used as constructors, super() or this() must occur *first* in the constructor

Abstract Classes

- An abstract class cannot be instantiated
- May have methods without bodies that *must be overridden* by any (non-abstract) subclass

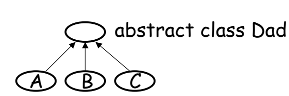
```
abstract class Puzzle {
   protected int state;
   public void scramble() {
      state = 978654321;
   }

   // Abstract methods (no code)
   abstract public int tile(int r, int c);
   abstract public void move(char d);
}
```

Abstract Classes

- An abstract class is an incomplete specification
  - Cannot be instantiated directly
  - Not all methods in an abstract class need to be abstract — allows code sharing
  - Abstract classes are part of the class hierarchy and the usual subtyping rules apply

Use of Abstract Classes



- Variables/methods common to a bunch of related subclasses can be declared once in Dad and inherited by all subclasses
- If subclass C wants to do something differently, it can override Dad's methods as needed

OOP Conclusions

- Object-oriented languages support data abstraction and code reuse
- Objects (instances of a class) can be created on demand by client without breaking abstraction
- Client can hold a reference to an object, but implementation is hidden from it
- User-defined types: class names are used as types of objects and references

- Key features of Object Oriented Programming (OOP)
  - Encapsulation: classes and access control
  - Inheritance: extending or changing the behavior of classes without rewriting them from scratch
  - Dynamic storage allocation
  - Access control: public/private/protected

Java Interfaces

- So far, we have mostly talked about interfaces *informally*, in the English sense of the word
  - An interface describes how a client interacts with a class
  - Method names, argument/return types, fields
- Java has a construct called interface which can be used formally for this purpose

Recall: A List Interface

```
public interface List {

    public void insert (Object element);

    public void delete (Object element);

    public boolean contains (Object element);

    public int size ();
}
```

- The interface specifies the methods without saying anything about the implementation
  - Matches idea of ADT (Abstract Data Type)
- Any class that *implements* List can be stored in a variable of type List

Another Java Interface Example

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

```
class IntPuzzle implements IPuzzle {

   public void scramble() {
      ... }
   public int tile(int r, int c) {
      ... }
   public boolean move(char d) {
      ... }
}
```

- Name of interface: IPuzzle
- A class implements this interface by implementing instance methods as specified in the interface
- The class may implement other methods as well

Interface Notes

- An interface is *not* a class!
  - Cannot be instantiated
  - Incomplete specification
- A class header must assert implements I for Java to recognize that the class implements interface I
- A class may implement several interfaces:
  - class X implements IPuzzle, IPod {
    ... }

Why an Interface Construct?

- Good software engineering
  - Can specify and enforce boundaries between different parts of a team project
- Can use interface as a type
  - Allows more generic code
  - Reduces code duplication

Example of Code Duplication

- Suppose we have two implementations of puzzles:
  - Class **IntPuzzle** uses an int to hold state
  - Class **ArrayPuzzle** uses an array to hold state
- Assume client wants to use both implementations
  - Perhaps for benchmarking both implementations to pick the best one
- Assume client has a display method to print out puzzles
  - What would the display method look like?

```
Class Client{
  IntPuzzle p1 = new IntPuzzle();
  ArrayPuzzle p2 = new ArrayPuzzle();
    ...display(p1)...display(p2)...

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

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

Code duplicated because IntPuzzle and ArrayPuzzle are different

Observation

- Two display methods are needed because IntPuzzle and ArrayPuzzle are different types, and parameter p must be one or the other
- But the code inside the two methods is identical!
  - Code relies only on the assumption that the object p has an instance method tile(int,int).
- Is there a way to avoid this code duplication?

One Solution: Abstract Classes

Puzzle code

```
abstract class Puzzle {
  abstract int tile(int r, int c);
  ...
}
class IntPuzzle extends Puzzle {
  public int tile(int r, int c) {...}
  ...
}
class ArrayPuzzle extends Puzzle {
  public int tile(int r, int c) {...}
  ...
}
```

Client code

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

Another Solution: Interfaces

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));
}}
```

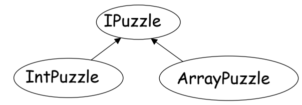
Extending vs. Implementing

- A class can extend just one superclass
  - Multiple inheritance can cause conflicts
  - Example: Which of 2 inherited methods to use when both have identical signatures?
- But a class can implement multiple interfaces
  - Multiple interfaces don't conflict because there are *no* implementations
- To share code between two classes
  - Put shared code in a common superclass
  - Interfaces *cannot* contain code

More on Interfaces

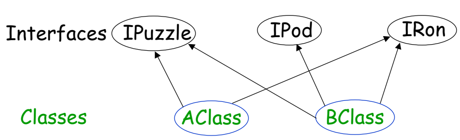
- Interface methods
  - Interface methods are implicitly public and abstract
  - No static methods are allowed in interfaces
- Interface constants
  - Interface constants are public, static, and final
  - Can inherit multiple versions of constants
    - Compiler detects this
    - When this occurs, fully qualified names are required

Interface Types



- Interface names can be used in type declarations
  - IPuzzle p1, p2;
- A class that implements the interface is a subtype of the interface type
  - IntPuzzle and ArrayPuzzle are *subtypes* of IPuzzle
  - IPuzzle is a *supertype* of IntPuzzle and ArrayPuzzle

Java Type Hierarchy



- Unlike Java classes, Java types do *not* form a tree!
  - A class may implement several interfaces
  - An interface may be implemented by several classes
- The type hierarchy does form a *dag* (directed acyclic graph)

Static Type vs. Dynamic Type

- Every variable (more generally, every expression that denotes some kind of data) has a static* or compile-time type
  - Derived from declarations – you can see it
  - Known at compile time, without running the program
  - Does not change
- Every object ever created also has a dynamic or runtime type
  - Obtained when the object is created using **new**
  - Not known at compile time – you can't see it

* Warning! No relation to Java keyword **static**

Example

```
int i = 3, j = 4;
Object x = new Integer(i+3*j-1);
System.out.println(x.toString());
```

- Static type of the variables i, j and the expression i+3*j-1 is int
- Static type of the variable x is Object
- Static type of the expression new Integer(i+3*j-1) is Integer
- Static type of the expression x.toString() is String (because toString() is declared in the class Object to have return type String)
- Dynamic type of the object created by the execution of new Integer(i+3*j-1) is Integer
  - It's dynamic type is still Integer even when it's held in variable x

Upcasting and Downcasting

- Applies to *reference types* (non-primitive types) only
- Used to assign the value of an expression of one (static) type to a variable of another (static) type
  - upcasting: subtype → supertype
  - downcasting: supertype → subtype
- Note that the dynamic type does not change!
- A crucial invariant:

> If the value of an expression *E* is an object *O* then the dynamic type of *O* is a subtype of the static type of *E*

Upcasting

- Example of upcasting:

```
Object x = new Integer(13);
```

  - Static type of expression on rhs is Integer
  - Static type of variable x on lhs is Object
  - Integer is a subtype of Object, so this is an upcast

- Static type of expression on rhs must be a subtype of static type of variable on lhs – compiler checks this

- Upcasting is always type correct – preserves the invariant automatically

Downcasting

- Example of downcasting:

```
Integer x = (Integer)y;
```

  - Static type of y is Object (say)
  - Static type of x is Integer
  - Static type of expression (Integer)y is Integer
  - Integer is a subtype of Object, so this is a downcast
- In any downcast, dynamic type of object must be a subtype of static type of cast expression
- Implies that a runtime check is needed to maintain invariant (and only time it is needed)
  - ClassCastException if failure

Is the Runtime Check Necessary?

- Yes, because dynamic type of object may not be known at compile time

```
void bar() {
  foo(new Integer(13));
}            String("x")

void foo(Object y) {
  int z = ((Integer)y).intValue();
  ...
}
```