Interfaces & Java Types

Lecture 10
CS211 – Fall 2005

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));
}}
```

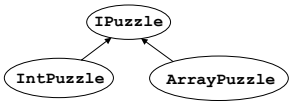
Extending vs. Implementing

- 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

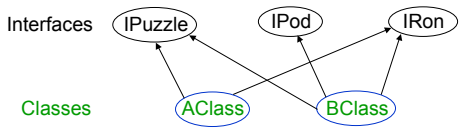
- Interface methods
  - Interface methods are implicitly **`public`** and **`abstract`**
  - No **`static`** methods are allowed in interfaces
- Interface constants
  - Interface constants are **`public`**, **`static`**, and **`final`**
  - Can inherit multiple versions of constants
    - Compiler detects 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`**

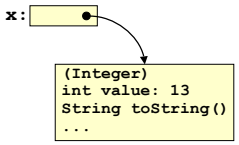
Example

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

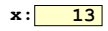
- static type of the variables **`i,j`** and the expression **`i+3*j-1`** is **`int`**
- static type of the variable **`x`** and the expression **`new Integer(i+3*j-1)`** is **`Integer`**
- static type of the expression **`x.toString()`** is **`String`** (because **`toString()`** is declared in the class **`Integer`** to have return type **`String`**)
- dynamic type of the object created by the execution of **`new Integer(i+3*j-1)`** is **`Integer`**

Reference vs. Primitive Types

- Reference types
  - Classes, interfaces, arrays
  - E.g.: **`Integer`**



- Primitive types
  - int, long, short, byte, boolean, char, float, double



Why Both int and Integer?

- Some data structures work only with reference types (`HashMap`, `ArrayList`, `Stack`,...)
- Primitive types are more efficient
  - `for (int i = 0; i < n; i++) {...}`

Upcasting and Downcasting

- Applies to *reference 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();
  ...
}
```

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`, respectively
- Static type of left-hand side variables is `IPuzzle`
- Rhs static types are subtypes of lhs static type, so this is OK

Why Upcasting?

- Subtyping and upcasting can be used to avoid code duplication
- Back to puzzle example: you and client agree on interface **`IPuzzle`**

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

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)

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));
}}
```

- At compile-time
  - Check that the static type of **`p`** (namely **`IPuzzle`**) has a **`tile`** method with the right type signature? No ⇒ error
- 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

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 of subtypes of **`IPuzzle`**

Java `instanceof`

- Example:
  `if (p instanceof IntPuzzle) {...}`
  - True if dynamic type of `p` is a subtype of `IntPuzzle`
  - Usually 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");
}
```

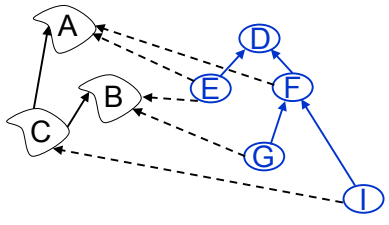
Subinterfaces

- 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

```
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




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

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