Interfaces & Types

Lecture 4
CS2110 – Fall 2008

Interfaces

- What is an interface? Informally, it is a specification of how an agent interacts with the outside world.
- Java has a construct called interface which is used formally for this purpose.
  - An interface describes how a class interacts with its clients.
  - Method names, argument/return types, fields.

Java interface

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

```java
class IntPuzzle implements IPuzzle {
    public void scramble() {...}
    public int tile(int r, int c) {...}
    public boolean move(char d) {...}
}
```

Notes

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

Why an interface construct?

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

Why an interface construct?

- Lots of examples in Java:
  ```java
  Map<String, Command> h = new HashMap<String, Command>();
  List<Object> t = new ArrayList<Object>();
  Set<Integer> s = new HashSet<Integer>();
  ```
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

• Say the client wants to use both implementations
  – perhaps for benchmarking both implementations to pick the best one
  – client code has a display method to print out puzzles

• What would the display method look like?

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

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) {...}
  ...
}

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

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) {...}
  ...
}

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));
}

• 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
• Unlike classes, types do not form a tree!
  – a class may implement several interfaces
  – an interface may be implemented by several classes

Extending a Class vs Implementing an Interface

• A class can
  – implement many interfaces, but
  – extend only one class

• To share code between two classes
  – put shared code in a common superclass
  – interfaces cannot contain code

Static vs Dynamic Types

• 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 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
  (Hashtable, Vector, 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

• A crucial invariant:
  If during execution, an expression \( E \) is ever evaluated and its value is an object \( O \), then the dynamic type of \( O \) is a subtype of the static type of \( E \).

Upcasting

• Example of upcasting:
  ```java
  Object x = new Integer(13);
  ```
  – static type of expression on rhs is \( \text{Integer} \)
  – static type of variable \( x \) on lhs is \( \text{Object} \)
  – \( \text{Integer} \) is a subtype of \( \text{Object} \), 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:
  ```java
  Integer x = (Integer)y;
  ```
  – static type of \( y \) is \( \text{Object} \) (say)
  – static type of \( x \) is \( \text{Integer} \)
  – static type of expression \( (\text{Integer})y \) is \( \text{Integer} \)
  – \( \text{Integer} \) is a subtype of \( \text{Object} \), so this is a downcast

• In any downcast, dynamic type of object must be a subtype of static type of cast expression
• runtime check, \( \text{ClassCastException} \) if failure
• needed to maintain invariant (and only time it is needed)

Is the Runtime Check Necessary?

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

```java
void bar() {
  foo(new Integer(13));
  }
  String("x")
void foo(Object y) {
  int z = ((Integer)y).intValue();
  ...
}
```

Upcasting with Interfaces

• Java allows up-casting:
  ```java
  IPuzzle p1 = new ArrayPuzzle();
  IPuzzle p2 = new IntPuzzle();
  ```
• Static types of right-hand side expressions are \( \text{ArrayPuzzle} \) and \( \text{IntPuzzle} \), resp.
• Static type of left-hand side variables is \( \text{IPuzzle} \)
• 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
• Puzzle example: you and client agree on interface \( \text{IPuzzle} \)
  ```java
  interface IPuzzle {
    void scramble();
    int tile(int r, int c);
    boolean move(char d);
  }
  ```
Solution

```java
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) {...}
    ...
}
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

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

- Which `tile` method is invoked?
  - depends on dynamic type of object `p` (IntPuzzle or ArrayPuzzle)
  - we don't know what it 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

- Compile-time check: does the static type of `p` (namely `IPuzzle`) have a `tile` method with the right type signature? If not → error
- Runtime: go to 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 on Casting

- Up- and downcasting merely allow the object to be viewed at compile time as a different static type
- Important: when you do a cast, either up or down, nothing changes
  - not the dynamic type of the object
  - not the static type of the expression

Another Use of Upcasting

Heterogeneous Data Structures

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

Java `instanceof`

- Example:
  ```java
  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`

```java
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

```java
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");
        }
    }
}
```

bad

good

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

```java
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

```java
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 modern programming languages
  – inheritance and interfaces are two methods for creating subtype relationships