CS 213 – Software Methodology
Spring 2017

Sesh Venugopal

Lecture 8 – Feb 9

Interfaces – Part 3

(Interface Polymorphism)

Using Interfaces: As a Front for Different Implementations (Plug and Play)

Stack structure

```
package util;

public class Stack<T> {
   private ArrayList<T> items;
   public Stack() {...}
   public void push(T t) {...}
   ...
}
```

Stack client

```
package apps;
import util.*;
public class SomeApp {
   ...
   Stack<String> stk =
       new Stack<String>();
   stk.push("stuff");
   ...
}
```

Using Interfaces: As a Front for Different Implementations (Plug and Play)

The `util` group wants to provide an alternative stack implementation that uses a linked list instead of an `ArrayList`.
In the process, it changes the name of the push method:

```
package util;

public class LLStack<T> {
   private Node<T> items;
   public LLStack() {...}
   public void llpush(T t) {...}
   ...
}
```

The client needs to make appropriate changes in the code in order to use the LL alternative:

```
package apps;
import util.*;
public class SomeApp {
   ...
   LLStack<String> stk =
       new LLStack<String>();
   stk.llpush("stuff");
   ...
}
```

To switch between alternatives, client has to make several changes.
Functionality (WHAT can be done - push) bleeds into implementation (HOW it can be done – ArrayList/Linked List) in the push/llpush methods.

Stack Alternatives: Better solution

Stack interface

```
package util;

public interface Stack<T> {
   void push(T t);
   T pop();
   ...
}
```

ArrayList version

```
package util;

public class ALStack<T>
implements Stack<T> {
   private ArrayList<T> items;
   public ALStack() {...}
   public void push(T t) {...}
   public T pop() {...}
   ...
}
```

Linked List version

```
package util;

public class LLStack<T>
implements Stack<T> {
   private Node<T> items;
   public LLStack() {...}
   public void push(T t) {...}
   public T pop() {...}
   ...
}
```

Stack Alternatives: Better solution

Stack client

```
package apps;

public class SomeApp {
   ...
   Stack<String> stk =
       new ALStack<String>();
   stk.push("stuff");
   ...
}
```

Use interface `Stack` for static type

To use other stack, only one change – in **new**

```
package apps;

public class SomeApp {
   ...
   Stack<String> stk =
       new LLStack<String>();
   stk.push("stuff");
   ...
}
```

Interfaces as a Front for Different Implementations – Example 2

In an application that does stuff with lists, there is a choice of what kind of list to use:

`ArrayList` used, statically typed to `ArrayList:`

```
ArrayList list = new ArrayList( );
. . .
list.<ArrayList method>(…)
. . .
```

OR

`ArrayList` used, statically typed to `List` (interface)

```
List list = new ArrayList( );
. . .
list.<List method>(. . .)
. . .
```

Interfaces as a Front for Different Implementations – Example 2

Consider later switching to a different implementation of a list, say `LinkedList`. The `LinkedList` class also implements the `List` interface.

In the version where `list` is statically typed to `ArrayList`:

```
LinkedList                   LinkedList
ArrayList list = new ArrayList( );
. . .
list.<ArrayList method>(…)
. . .              ?
```

What if this method is not in the `LinkedList` class?

Need to check *all* places where a `list`.<method>(…) is called. Then keep it as it is (same functionality is in `LinkedList`), or change it to an equivalent `LinkedList` method (if one exists), and if not, somehow devise equivalent code.

Interfaces as a Front for Different Implementations – Example 2

Consider later switching to a different implementation of a list, say `LinkedList`. The `LinkedList` class also implements the `List` interface.

In the version where `list` is statically typed to `ArrayList`:

```
                   LinkedList
List list = new ~~ArrayList~~( );
. . .
list.<List method>(…)
. . .
```

Just replace `new ArrayList()` with `new LinkedList()`
No other changes needed

Using an interface type to switch implementations is called interface polymorphism

Using Interfaces: As a Workaround for Multiple Inheritance

```
public class Phone {
   public void makeCall(...) {...}
   public void addContact(...) {...}
   ...
}
```

```
public class MusicPlayer {
   public Tune getTune(...) {...}
   public void playTune(...) {...}
   ...
}
```

Want a class to implement a device that is both a phone and a music player:

```
public class SmartPhone
extends Phone, MusicPlayer {
   public void makeCall(...) {...}
   public void addContact(...) {...}
   public Tune getTune(...) {...}
   public void playTune(...) {...}
   ...
}
```

Can’t extend more than one class!

Using Interfaces: As a Workaround for Multiple Inheritance

```
public class Phone {
   public void
      makeCall(...) {...}
   public void
      addContact(...) {...}
   ...
}
```

```
public class MusicPlayer {
   public Tune
      getTune(...) {...}
   public void
      playTune(...) {...}
   ...
}
```

Workaround is to define at least one of the types as an interface:

```
public interface MusicPlayer {
   Tune getTune(...);
   void playTune(...);
   ...
}
```

Drawback is `getTune` and `playTune` will have to be re-implemented in `SmartPhone` instead of being reused from `MusicPlayer`

```
public class SmartPhone
  extends Phone
  implements MusicPlayer {
   public void makeCall(...) {...}
   public void addContact(...) {...}
   public Tune getTune(...) {...}
   public void playTune(...) {...}
   ...
}
```

Summary: Some uses of Interfaces

- To define ("prescribe") one or more special roles needed by separately built algorithm/ functionality (e.g. 3-outcome comparison in binary search). Clients can support these roles by implementing the interface
- To support plug-and-play of different implementations to the same interface
- To work around multiple inheritance

Polymorphism with super/sub classes

```
public class Point {
   int x,y;
   . . .
   public String toString( ) {
     return x + “,” + y;
   }
}

public class ColoredPoint
extends Point {
   String color;
   . . .
   public String toString( ) {
     super.toString() + “,” + color;
   }
}
```

```
// client code
Point[] pts = new Point[n];
// fill pts with a mix of Point
// and ColoredPoint objects
pts[0] = new Point(2,3);
pts[1] = new ColoredPoint(3,4,”red”);
...
for (int i; i < n; i++) {
     System.out.println(pts[i]);
}
```

Polymorphism!

Depending on whether the run time object is `Point` or `ColoredPoint`, the appropriate `toString` method is called (dynamic binding)

“Polymorphism” because `pts[i]` automatically takes a different “shape”, either `Point` or `ColoredPoint`, at runtime