CS 213 – Software Methodology

Sesh Venugopal

Interfaces - Uses

Interface Uses:

1. To Have Classes Conform to a Specific Role Used in External Context

Classes – Conform to Specific External Role

Often,
a specialized role needs to be specified
for some classes in an application (e.g. comparing for ==, >, <),
and given a type name (.e.g. `Comparable, EventHandler`)

The type name is the interface name,
and the role is the set of interface methods.

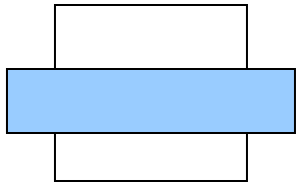
You can think of an interface as
a filter that is overlaid on a class.

Depending on the context,
the class can be fully itself (class type)
or can adopt a subset, specialized role (interface type)

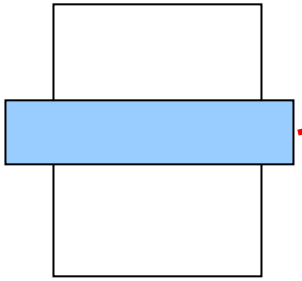
Specialized Role For Classes

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

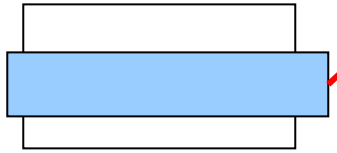
class X implements Comparable<X>



class Y implements Comparable<Y>



class Z implements Comparable<Z>



methodM will admit any object, so long as it is Comparable, and it knows the admitted object ONLY as Comparable – that is, the filter is blind to all other aspects of the object type (X, or Y, or Z) but the Comparable part

class U

```
static
<T extends Comparable<T>>
void methodM(T c) {
   ...
}
```

The implementor of methodM in class U may use the compareTo method on the parameter object c, without knowing anything about the argument except that it will be guaranteed to implement compareTo

Interface Uses:

2. To assign a single type that can stand for common functionality in related classes
(when these classes are not in an inheritance hierarchy)

Type for Classes with Common Behavior

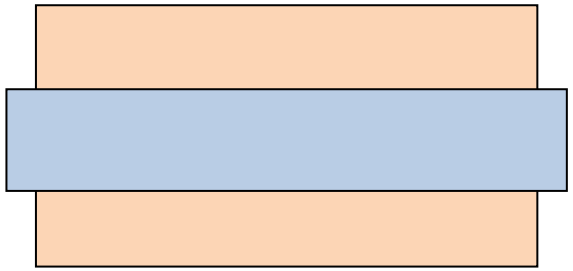
Zebras, Horses and Donkeys can all trot, gallop, and snap (common behavior)

In a simulation with many instances of each, you may want to evoke one or more of these behaviors in a randomly selected instance or group, without regard to what exact specimen is targeted – grouping these behaviors under a type meets this need

```
public interface Equine {
   void trot();
   void gallop();
   void snap();
 }
```

`class Zebra implements Equine`

`class Horse implements Equine`



`class Donkey implements Equine`

Polymorphism using interface type

A collection (e.g. `ArrayList`) might have a combination of zebras, donkeys and horses

```
ArrayList<Equine> equines = new ArrayList<>();
equines.add(new Zebra());
equines.add(new Horse());
...
```

Now you can apply any of the common behaviors to instances of the collection, without regard to the actual type of animal (no need to check what actual type it is):

```
for (Equine eq: equines) {

  eq.trot();
   ...
}
```

actual behavior is executed on runtime instance

This is polymorphism via an interface type – common behavior executed on objects with same interface (static) type, but the way the behavior is executed is automatically determined by binding to the run time type ("shape" of object changes automatically, hence poly "morph" ism.)

Interface Uses:

3. To Set Up an Invariant Front for Different Implementations of a Class

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");
   ...
}
```

(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() {...}
   ...
}
```

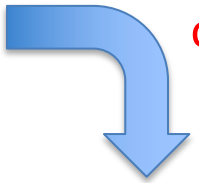
Stack Alternatives: Better solution

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");
    ...
}
```

Plug and Play – 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>(. . .)
. . .
```

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.

Example 2

But, in the version where `list` is statically typed to `List`:

```
                    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 a kind of interface polymorphism

Interface Uses:

4. As a workaround for multiple inheritance

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!

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(...) {...}
   ...
}
```