Programming Mechanisms That Can Solve Design Problems

Week 2

Common Design Challenges

- How can I define a set of behaviors along with their implementation?
- How can I define a general set of behaviors without worrying about their implementation?
- How can I (re)implement a set of new/old behaviors while taking advantage of existing behaviors
  - Need compatibility with existing code
    - how can I define a new set of behaviors without turning the whole world upside down?
- How can I implement a broad set of behaviors without being responsible for defining the behaviors themselves.
  - Also, how can I define a class and its behaviors without implementing all of them myself?
- How can “bundle” up my services and make them available to other programs?

How can I define a set of behaviors along with their implementation?

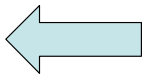
Define a class.

```
class Complete {
        int x;
        Complete() { x  = 100; }
        Value() { return x; }
}
```

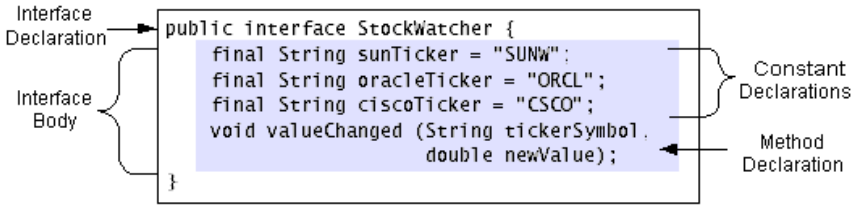
How can I define a general set of behaviors without worrying about their implementation?

- Use an interface

Definition: An *interface* is a named collection of method definitions (without implementations). An interface can also declare constants

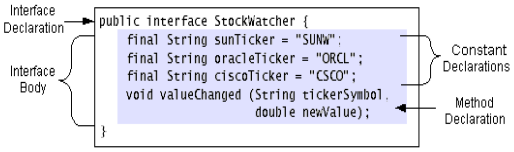


"what is must do, but not how it does it!"



How can I implement a broad set of behaviors without being responsible for defining the behaviors themselves.

- Use “implements” with interface.




```
class PalmOrganizer implements StockWatcher
        PalmOrganizer() {…}
        void valueChanged(String tickerSymbol,
                          double newValue) {
                        PDA.beepBeep(tickerSymbol); }
        void updateCalendar() {…} // other cool methods
}
```

Interface Properties

- An interface name defines a new reference data type.
- You can use interface names anywhere you can use any other data type name.
- Only an instance of a class that implements the interface can be assigned to a reference variable whose type is an interface name.

```
StockWatcher sw = new StockWatcher(); // legal??
```

Stack Example

- Interface Stack
  - LIFO structure.
- Implementation FixedStack
  - Fixed stack
  - Simple, but limited
- Implementation DynamicStack
  - Dynamic allocation.
  - Unlimited, but more complicated
  - Two “tricks”
- Note javadoc behavior!

The trouble with Interfaces

- Over time, systems change.
  - Suppose we want to allow stack clients to query the state of the stack?
    - isEmpty()? isFull()?, etc…
- If you change your interface specification, then all implementing code must change.
- Two issues
  - Who implements?
  - What should the implementations do in light of the change?
    - Consider: public boolean isEmpty()
    - Lots of busy work (typing)
      - $$

How can I implement a new set of new/old behaviors while taking advantage of existing behaviors

- While maintaining compatibility with existing clients
- Or, how can I define a new set of behaviors without turning the whole world upside down?

- Solution: Inheritance
  - **Definition:** A *subclass* is a class that extends another class. A subclass ***inherits*** state and behavior from all of its ancestors. The term "superclass" refers to a class's direct ancestor as well as to all of its ascendant classes

What does Inheritance Allow/Deny?

- A subclass inherits variables and methods from its superclass and all of its ancestors. The subclass can use these members as is, or it can hide the member variables or override the methods.
  - A subclass cannot override methods that are declared final in the superclass (by definition, final methods cannot be overridden).
    - If you attempt to override a final method, the compiler displays an error message similar to the following and refuses to compile the program.

```
class ancestor {
   public void bar() { … }
  final public void foo();
}

class descendant extends ancestor {
   public void bar() {…}  // ok? Yes or no?
  public void foo() {…} // ok?? Yes or no?
}
```

Stack Example. Redux.

- Use inheritance, not interface.
- Base class implements “some kind of stack”
  - How it works? We don’t care.
- Subclasses take care of richer services as necessary.
  - Eg. How many were pushed, popped, etc.

Something really fancy?

- How would we define yet a new class that also counts the number of items that have been popped?
- And then, an even better one that tells us if pushes == pops?