Inheritance vs. delegation

Outline for Week 13

Delegation—where one object passes a message on to another object—can often achieve the same effect as inheritance. Let's look at an example.

Consider the **`java.util.Stack`** class. How many operations does it have?

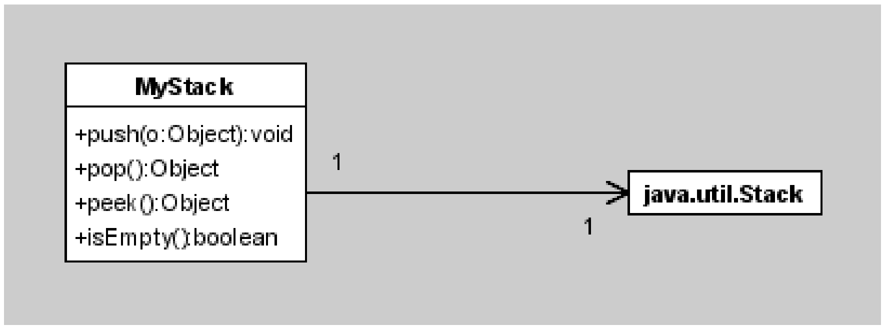
Suppose in a program you want a "pure" stack class—one that can only be manipulated via **`push(…)`** and **`pop()`**.

Why would you want such a class, when Java already gives you that and more?

What is the "simplest" way to get a pure **`Stack`** class?

Or you could create **`Stack`** class "from scratch." What's wrong with doing this?

Another option is to create your own **`Stack`** class, but have it *include* a **`java.util.Stack`**.



What is the name for the approach are we using here?

Here's what this class might look like.

```
public class MyStack
{
  private java.util.Stack stack;
  public MyStack(){stack = new java.util.Stack();}
  public void push(Object o) { stack.push(o); }
  public Object pop() { return stack.pop(); }
  public object peek() { return stack.peek(); }
  public boolean isEmpty(){return stack.empty();}
}
```

Delegation is particularly useful where objects might need to "change state"—think of a student becoming an employee. Both Student and Employee can delegate to Person.

Exercise: Delegation in a sorted list

This exercise is an example of creating a sorted **`ArrayList`** of **`String`**s by delegating to Java's **`ArrayList`** class. Every time an element is added to the list, the **`sort`** method of **`Collections`** is called.

This exercise asks you to fill in the blanks so that the list stays sorted.

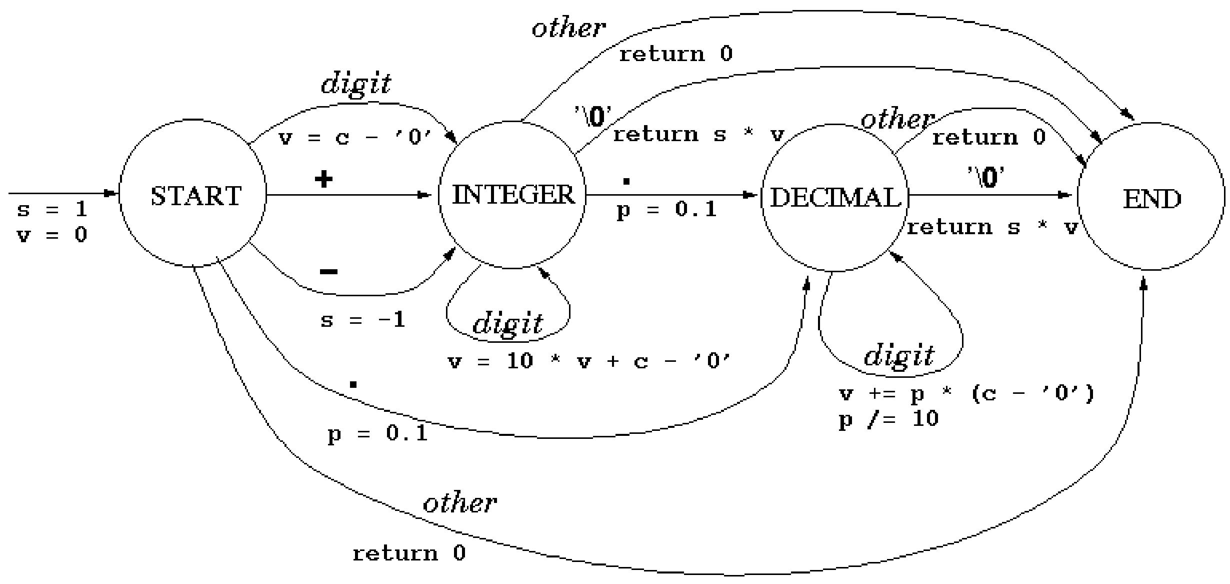
State pattern

We've been talking about bad uses of **case** statements in programs. What is one example?

Another way in which case statements are sometimes used is to implement finite-state machines.

An example: Horner's Rule

A finite-state machine can be used to convert an ASCII string of characters representing a real number to its actual numerical value.



Input symbols are written in **bold** above the transitions.

Symbol classes (such as *digit*) are written in *italic*.

Actions performed by the program are in `courier` below the transitions.

The variable `c` denotes the input character.

The letters shown in the FSM stand for the following:

c –	current character	**v** –	value of the number
s –	sign of the number	**p** –	power

Note that this FSM assumes that the string contains a valid floating-point number that

- starts with an optional + or –,
- has at least one digit, an optional decimal point,
- and any number (including 0) of digits before and after the decimal point.

A value of 0 is returned if an invalid string is encountered.

Table form of FSM:

State/Input	**+ or –**	**.**	**digit**	**other**
START	**INTEGER**	**DECIMAL**	**INTEGER**	**ERROR**
INTEGER	**ERROR**	**DECIMAL**	**INTEGER**	**END**
DECIMAL	**ERROR**	**ERROR**	**DECIMAL**	**END**

Using switch statements, this FSM can be coded as follows:

```
public class Parser {
   static double toDouble(String s) {
      double sign = 1;  // sign of number (either 1 or −1)
      double value = 0;  // current value of the number
      double power = 0.1;  // current power of 10 for
                                          // digits after decimal point
      int i = 0;
      final int START = 0;
      final int INTEGER = 1;
      final int DECIMAL = 2;
      final int ERROR = 3;
      int state = START;
      char ch;  //current character in string

      while (state != ERROR && i < s.length()) {
         ch = s.charAt(i++);
         switch (state) {
            case START: if (ch == '.')
                  state = DECIMAL;
               else if (ch == '-') {
                  sign = -1.0;
                  state = INTEGER;
               }
               else if (ch == '+')
                  state = INTEGER;
               else if (Character.isDigit(ch)) {
                  value = ch - '0';
                  state = INTEGER;
               }
               else
                  state = ERROR;
               break;
            case INTEGER: if (ch == '.')
                  state = DECIMAL;
               else if (Character.isDigit(ch))
                  value = 10.0 * value + (ch - '0');
               else {
                  value = 0.0;
                  state = ERROR;
               }
               break;
            case DECIMAL: if (Character.isDigit(ch)) {
                  value += power * (ch - '0');
                  power /= 10.0;
               }
               else {
                  value = 0.0;
                  state = ERROR;
               }
               break;
            default: System.out.println("Invalid state: " + state);
         }
      }
      return sign * value;
   }

   public static void main(String[] args) {
      if (args.length == 1)
         System.out.println(toDouble(args[0]));
   }
}
```

This FSM can be represented more elegantly by the State pattern.

How can we code State in a more o-o fashion? *Hint:* We can make **`State`** an interface! Each state will implement this interface.

Horner's Rule: To use the State pattern for Horner's rule, the first step is to define a **`State`** interface. Consider the table form of the FSM.

- The rows of the table represent the different states.
- The columns of the table represent the different *behaviors* of each state.

Therefore, what *methods* should be defined in the **`State`** interface? Well, what do we need to test for each state?

Submit your **`State`** interface here.

```
public interface State {
    void onPoint();
    ____________________;
    ____________________;
    ____________________;
    void onOther();
}
```

How should the **`State`**s be defined?

class ________ **implements** ______ **{ … }**

class ________ **implements** ______ **{ … }**

class ________ **implements** ______ **{ … }**

In Java, we can define a class within another class. This is called an *inner class*.

Thus, our **`State`**s can be defined as inner classes of **`Parser`**.

Here is the code for the **`Parser`** class, minus its inner classes:

```
public class Parser {
    private final State start = new Start();
    private final State integer = new Integr();
    private final State decimal = new Decimal();
    private State state = start;
    double sign = 1; // sign of number (either 1 or -1)
    double value = 0; // current value of the number
    double power = 0.1; // current power of 10 for
                         // digits after decimal point
    char ch; //current character in string

    double toDouble(String s) {
       int i = 0;
       while (i < s.length()) {
          ch = s.charAt(i++);
          if (ch == '.') state.onPoint();
          else if (ch == '+') state.onPlus();
          else if (ch == '-') state.onMinus();
          else if (Character.isDigit(ch))state.onDigit();
          else state.onOther();
       }
       return sign * value;
     }

     public static void main(String[] args) {
      System.out.println(new Parser().toDouble("-
914.334"));
     }
}
```

Exercise: Choose one method to implement in all three classes. Submit your code here.

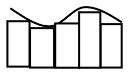
```
        void onMinus();
        void onPlus();
        void onDigit();
        void onOther();
```

If an illegal character is found, throw a `NumberFormatException`.

Strategy pattern

A related pattern is Strategy. This pattern helps when you need to choose an algorithm for a task depending on some "parameter" of the situation.

For example, consider quadrature (numerical integration) again. Each time you calculate the area of a region, you need to know what the function is that you are calculating the region underneath.



Or consider converting different file formats, e.g., .jpeg, .gif, .eps.

You *could* write a case statement whenever you needed to invoke one of the algorithms. Is this a good idea?

Consider extensibility and maintainability.

But suppose there is only one case statement. Is it OK then?

Another situation might be where you are manipulating several geometric shapes, e.g., circles, squares, and composites of circles and squares. You need to—

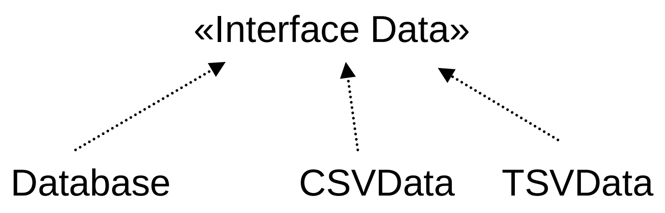
- draw the shapes on a display
- move them to a different location
- rotate them by a certain number of degrees.

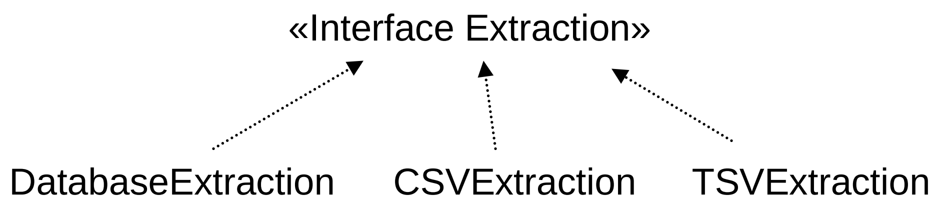
These tasks will be performed differently for each shape. You *could* use a case statement everywhere you need to make the decision. But that violates the DRY pattern.

The Strategy pattern allows you to make the decision once when you begin to handle the shapes, and all of the other actions are performed accordingly.

Exercise: Another common situation is when you are working with various kinds of files. You need to open, close, and access them differently depending on the file type.

Our example looks like this.





Fill in the blanks to complete the pattern.

State vs. Strategy

A definition of Strategy (from *Head-First Design Patterns*) is,

> The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Strategy allows clients to change algorithms at run time by using a different strategy object. This basically lets them appear to change class at run time.

Hmmm … that's interesting. Strategy lets objects appear to change class. Isn't that what State does?

What are the differences between Strategy and State?

- State requires objects to "change state"; that's the point of the pattern. In Strategy,
- The State pattern deals with *how* you switch between different states/implementations, while the Strategy pattern deals with *using* different implementations to implement different algorithms.
- In State, the behavior that's invoked typically depends both on the current state and an input. In Strategy,
- Martin Fowler: "*We encapsulate each algorithm into a class in strategy pattern, but we encapsulate each state into a class in state pattern*.

The Visitor pattern

Remember our discussion of overloading vs. overriding, from Week 9? At that time, we said,

In summary, the compiler decides which overloaded method to call by looking at

- the current type of the object being sent the message and
- the declared types of the arguments to the method call.

The method is chosen at runtime by dynamic method invocation using the actual value of the object being sent the message.

The actual classes of the arguments to the method call do not play a role.

This is very different from a language like CLOS, which uses the actual types of the arguments to decide which method to execute.

Suppose we *did* want the classes of the arguments to be used to determine which method to call.

My favorite example is "double-dispatching" in arithmetic expressions.

- If you add an integer and a floating-point number, what type should the result be?

- Assuming you have a Fraction class, if you add an integer and a fraction, what type should the result be?

- If you add a floating-point number and a complex number, what type should the result be?

Help answer these questions by filling in this table.

Should *either* the floating-point or complex number be able to be the receiver? Should either be able to be the argument?

So, the method called should depend both on the class of the receiver and the class of the argument. How do we achieve this effect?

Let's say that we implement the Sum method in all numeric classes—Integer, Floating Point, Fraction, and Complex.

So, if we're performing an addition, we invoke the Sum method of the __________ class.

Now, this Sum method knows that what it does actually depends on the class of its argument. How does it achieve this effect?

This method, e.g., in the Complex class, is called something like, SumFromFloatingPoint.

- What does it do?

- What does it return?

OK, suppose that we have the four numeric classes mentioned above. How many Sum… methods do we need altogether?

What is the sequence of calls?

- **`mySum = myFloat.sum(myComplex)`**
    - **`return`**

Here is how Visitor is structured.

- Define an interface or abstract class Visitor.
- Visitor contains a **`visit()`** method, which is implemented in each subclass of Visitor. (In our example, these are the **`sumFrom`** methods.)
- These methods are invoked from subclasses of the Element hierarchy. Each one of these classes has an **`accept()`** method, which takes an object of the Visitor hierarchy as a parameter.
- Each descendant of the Element class implements **`accept()`** by calling the **`visit()`** method on the Visitor object it was passed, with **`this`** as the only parameter.
- To perform an operation, the client creates a Visitor object, and calls **`accept()`** on the Element object, passing the Visitor object.

The Visitor pattern can be used to avoid tight coupling, as Bob Martin explains.

Here is a similar example for you to complete.