MVC: Model, View, Controller

- A model is the state and brains of a system
  - In a game it's all the pieces and where they are
  - In a spreadsheet it's the data and the formulae
- The view is how we look at the model
  - Spread sheet has graphs, charts, cells, text, …
  - Game has board, number of opponents, hit-points, …
- When the model changes, the views reflect the changes
  - The model tells the views how/if it has changed
  - Model sends information to views OR
  - View asks model for information

MVC: interfaces and inheritance

- A model might have multiple views
  - Tell all the views "I've changed"
  - Who manages the views? This requires state: store views
  - Why can't we keep this state in an interface?
- See IModel and AbstractModel
  - One specifies behavior, the other provides default
  - Don't rewrite code if we don't have to, maintaining views will be the same for all models
- See IView and SimpleView
  - No default/shared view state/behavior: text and GUI

Does SimpleViewer know Model?

- What does the SimpleViewer know about its model?
  - If we look at code, is there any application-specific logic?
  - What if we wanted to play a game, start a new game?
- Control in MVC with SimpleViewer and IModel
  - Loading a file calls `initialize()`
  - Entering text calls `process()`
  - Model calls view with messages, errors, and complete update
- This isn't complete general, but it's pretty generic
  - For this input, here's the output

Pixmap Assignment

- Traditional "Last" CompSci 6 Assignment
  - Lots has been done for you
  - Mainly an exercise in working with 2 D info
- Not really MVC
  - Doesn't hurt to keep that model in mind, though
- Lots of GUI stuff
  - Graphical User Interface is not reall focus of this course
  - Just use what has been given
  - Become familiar with it by reading code, seeing results
- Feel free to experiment

Java Exceptions

- Many I/O operations can throw Exceptions
  - Code handles it for your
  - However, need to know what is going on
  - (Review pages in Chapter 2)
- Catching Exceptions
  - Use `try-catch` block

```
try {
   // statements that might generate exception
}
catch (Exception_type var) {
   // code that deals with exception
}
```

- Method can *pass on* responsibility for exception with `throws` clause

Stack: What problems does it solve?

- Stacks are used to avoid recursion, a stack can replace the implicit/actual stack of functions called recursively
- Stacks are used to evaluate arithmetic expressions, to implement compilers, to implement interpreters
  - The Java Virtual Machine (JVM) is a stack-based machine
  - Postscript is a stack-based language
  - Stacks are used to evaluate arithmetic expressions in many languages
- Small set of operations: LIFO or last in is first out access
  - Operations: push, pop, top, create, clear, size
  - More in postscript, e.g., swap, dup, rotate, …

Simple stack example

- `Stack` is part of java.util.Collections hierarchy
  - It's an OO abomination, extends Vector (like ArrayList)
    - Should be implemented using Vector
    - Doesn't model "is-a" inheritance
  - What does pop do? What does push do?

```
Stack s = new Stack();
s.push("panda");
s.push("grizzly");t o = s.pop();
s.push("brown");
System.out.println("size = " + s.size());
System.out.println(s.peek());
Objec
System.out.println(s.peek());
System.out.println(s.pop());
```

Implementation is very simple

- Extends Vector, so simply wraps Vector/ArrayList methods in better names
  - push==add, pop==remove
  - Note: code below for ArrayList, Vector is actually used.

```
public Object push(Object o){
    add(o);
    return o;
}
public Object pop(Object o){
    return remove(size()-1);
}
```

Uses rather than "is-a"

- Suppose there's a private ArrayList, myStorage
  - Doesn't extend Vector, simply *uses* Vector/ArrayList
  - Disadvantages of this approach?
    - Synchronization issues

```
public Object push(Object o){
    myStorage.add(o);
    return o;
}
public Object pop(Object o){
    return myStorage.remove(size()-1);
}
```

Postfix, prefix, and infix notation

- Postfix notation used in some HP calculators
  - No parentheses needed, precedence rules still respected

```
3 5 +          4 2 * 7 + 3 -          9 7 + *
```

  - Read expression
    - For number/operand: push
    - For operator: pop, pop, operate, push
- See *Postfix.java* for example code, key ideas:
  - Use `StringTokenizer`, handy tool for parsing
  - Note: Exceptions thrown, what are these?
- What about prefix and infix notations, advantages?

Exceptions

- Exceptions are *raised* or *thrown* in exceptional cases
  - Bad indexes, null pointers, illegal arguments, …
  - File not found, URL malformed, …
- Runtime exceptions aren't meant to be handled or *caught*
  - Bad index in array, don't try to handle this in code
  - Null pointer stops your program, don't code that way!
- Other exceptions must be caught or rethrown
  - See `FileNotFoundException` and `IOException` in Scanner class implementation
- RuntimeException `extends` Exception, `catch` not required

Prefix notation in action

- Scheme/LISP and other functional languages tend to use a prefix notation

```
(define (square x) (* x x))

(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

Postfix notation in action

- Practical example of use of stack abstraction
- Put operator after operands in expression
  - Use stack to evaluate
    - operand: push onto stack
    - operator: pop operands push result
- PostScript is a stack language mostly used for printing
  - drawing an “X” with two equivalent sets of code

```
%!
200 200 moveto
100 100 rlineto
200 300 moveto
100 –100 rlineto
stroke showpage
```

```
%!
100 –100 200 300 100 100 200 200
moveto rlineto moveto rlineto
stroke showpage
```