Undo

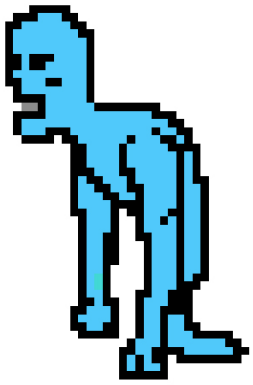
principles, concepts, and Java implementation

Undo Benefits

- Undo enables exploratory learning
  - "One of the key claims of direct manipulation is that users would learn primarily by trying manipulations of visual objects rather than by reading extensive manuals." [Olsen, p. 327]
  - try things you don't know the consequences of (without fear or commitment)
  - try alternative solutions (without fear or commitment)
- Undo lets you recover from errors
  - input errors (human) and interpretation errors (computer)
  - you can work quickly (without fear)
- Undo lets you evaluate modifications
  - fast do-undo-redo cycle to evaluate last change to document

Checkpointing

- A manual undo method
  - you save the current state so you can rollback later (if needed)
- Consider a video game …
  - You kill a monster
  - You save the game
  - You try to kill the next monster
  - You die
  - You reload the saved game
  - You try to kill the next monster
  - You kill the monster
  - You save the game



- Source code repositories are a type of checkpointing

Undo Design Choices

1. **Undoable Actions**: what actions should (or can) be undone?
2. **State restoration**: what part of UI is restored after undo?
3. **Granularity**: how much should be undone at a time?
4. **Scope:** is undo global, local, or someplace in between?

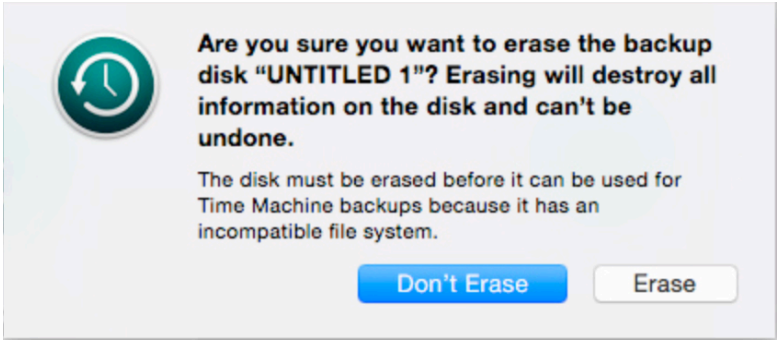
Undoable Actions

- Some actions may be omitted from undo
  - Change to selection? Window resizing? Scrollbar positioning?
- Some actions are destructive and not easily undone
  - e.g. quitting program with unsaved data, emptying trash
- Some actions can't be undone
  - e.g. printing

Undoable Actions: Suggestions

- All changes to document (i.e. the model) should be undoable
- Changes to the view, or the document's interface state, should be undoable if they are extremely tedious or require significant effort

- Ask for confirmation before doing a destructive action which cannot easily be undone



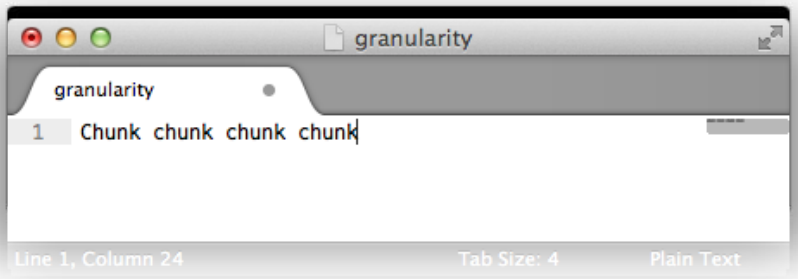
State Restoration

- What is the user interface state after an undo or redo?
  - e.g. highlight text, delete, undo … is text highlighted?
  - e.g. select file icon, delete, undo … is file icon highlighted?

- User interface state should be meaningful after undo/redo action
  - Change selection to object(s) changed as a result of undo/redo
  - Scroll to show selection, if necessary
  - Give focus to the control that is hosting the changed state
- These provide additional undo feedback

Granularity

- What defines one undoable “chunk”?
  - chunk is the conceptual change from one document state to another state
- Examples
  - MS Word → string delimited by any other command (bold, mouse click, autocorrect, etc…)
  - Sublime Text Editor → token delimited by whitespace
  - Google Write -> text since last save
  - IOS Mail → all text since key focus



Granularity: Drawing Example

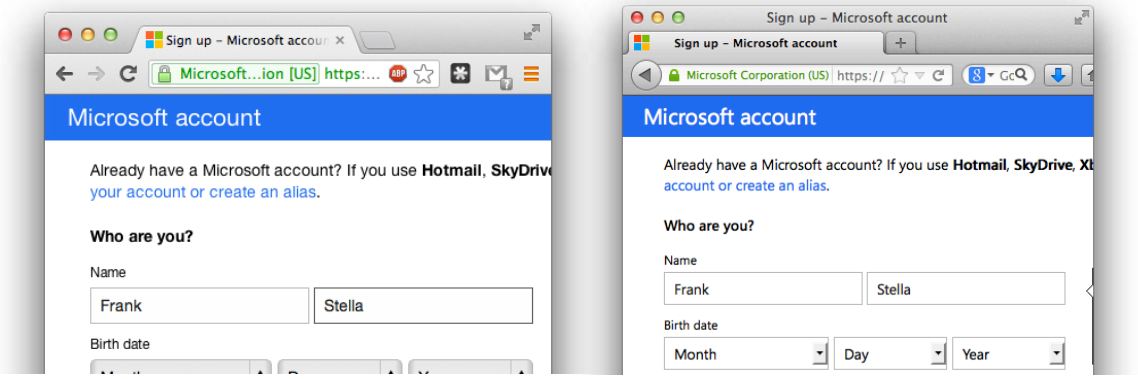
- MouseDown to start line
- MouseDrag to define line path
- MouseUp to end line
- MouseDown + MouseDrag + MouseUp = 1 chunk
  - "undo" should probably undo the entire line, not just a small delta in the mouse position during MouseDrags

Granularity: Suggestions

- Ignore direct manipulation intermediate states
  - Examples:
- Chunk all changes resulting from an interface event
  - Examples:
- Delimit on discrete input breaks
  - Examples:

Scope

- Where does undo happen?
  - System level?
  - Application level?
* - Document level?
  - Widget level?
- Example: undo form values in Firefox vs. Chrome



Implementing Undo

- Forward Undo
  - save complete baseline document state at some past point
  - save *change records* to transform baseline document into current document state
  - to undo last action, don't apply last *change record*
- Reverse Undo
  - save complete current document state
  - save reverse *change records* to return to previous state
  - to undo last action, apply last reverse *change records*

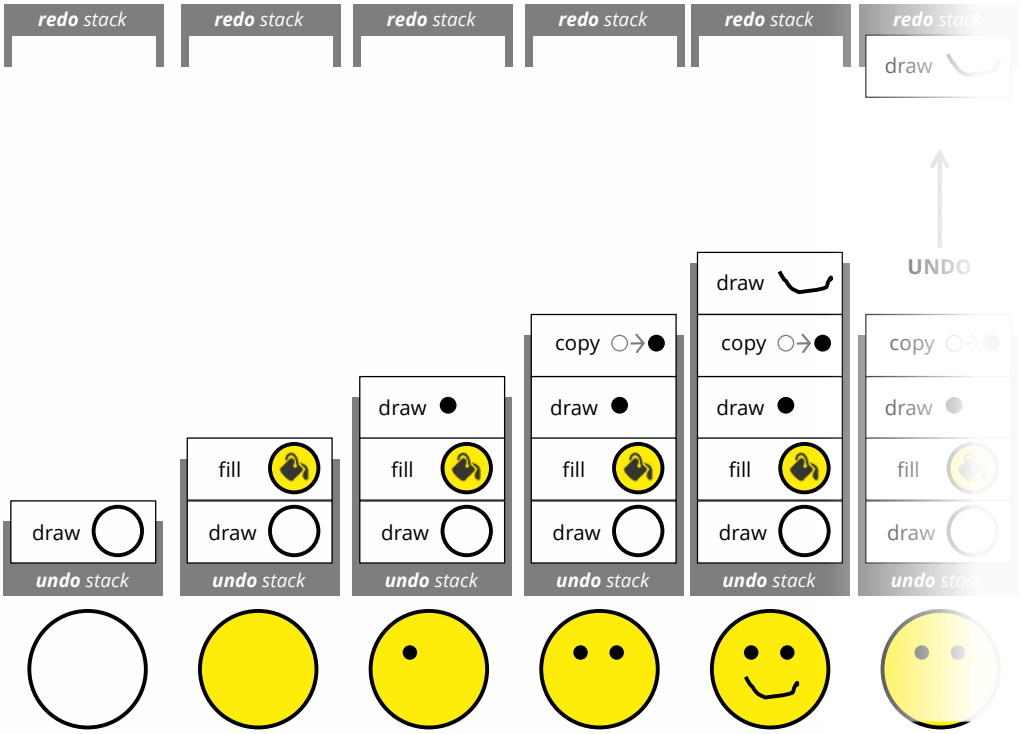
Change Record Implementation

- Option 1: **Memento** pattern
  - save snapshots of each document state
  - could be complete state or difference from “last” state
- Option 2: **Command** pattern
  - save commands to execute (or “un-execute”) to change state
- Java platform uses reverse undo with command pattern
  - but may need Memento to save states when “information is lost”

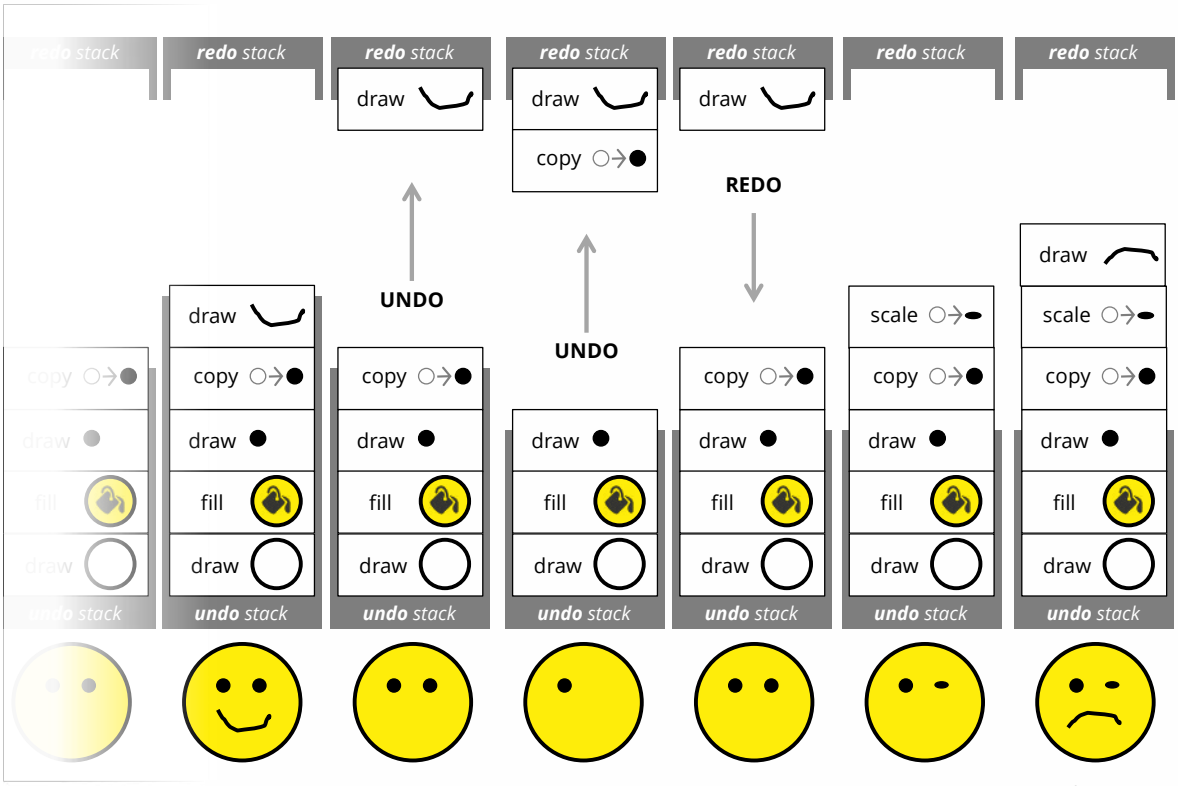
Reverse Undo Command Pattern

- User issues command
  - execute command to create new current document state
  - push command onto undo stack
  - clear redo stack
- Undo
  - pop command from undo stack and un-execute it to create new current document state (which is the previous state)
  - push command on redo stack
- Redo
  - pop command off redo stack and execute it to create new current document state
  - push command on undo stack

Undo and Redo Stacks



Undo and Redo Stacks



Example: Text Editor Undo/Redo Commands

- Available Commands:

```
insert(string, start)
delete(start, end)
bold(start, end)
normal(start, end)
```

<start>	Quick brown	`insert("Quick brown", 0)`
<command>	Quick **brown**	`bold(6, 10)`
<command>	Quick **brown** fox	`insert(" fox", 11)`
<undo>	Quick **brown**	`delete(11, 14)`
<undo>	Quick brown	`normal(6, 10)`
<redo>	Quick **brown**	`bold(6, 10)`
<command>	Quick **brown** dog	`insert(" dog", 11)`

Example: Text Editor Undo/Redo Commands

Command	Document	Undo Stack	Redo Stack
insert("Quick brown", 0)	Quick brown	delete(0, 10)	<empty>
bold(6, 10)	Quick **brown**	*normal(6, 10)* delete(0, 10)	<empty>
insert(" fox", 11)	Quick **brown** fox	*delete(11, 14)* normal(6, 10) delete(0, 10)	<empty>
undo	Quick **brown**	normal(6, 10) delete(0, 10)	*insert(" fox", 11)*
undo	Quick brown	delete(0, 10)	*bold(6, 10)* insert(" fox", 11)
redo	Quick **brown**	*normal(6, 10)* delete(0, 10)	insert(" fox, 11)
insert(" dog", 11)	Quick **brown** dog	*delete(11, 4)* normal(6, 10) delete(0, 10)	<empty>

Java Undo

- Java's undo functionality in `javax.swing.undo.*`
  - UndoManager keeps track of undo/redo command stacks
  - UndoableEdit interface is the command to execute (redo) or un-execute (undo)
- Usually put UndoManager in Model for document context

```
import javax.swing.undo.*;

// A simple model that is undoable
public class Model extends Observable {

   // Undo manager
   private UndoManager undoManager;

   ...
}
```


UndoableEdit in Model Setters

```
public void setValue(int v) {
   final int oldValue = value;
   final int newValue = v;

   // create undoable edit
   UndoableEdit undoableEdit = new AbstractUndoableEdit() {
      public void redo() {
         value = newValue; // the redo command
         notifyObservers();
      }
      public void undo() {
         value = oldValue; // the undo command
         notifyObservers();
      }
   };
   undoManager.addEdit(undoableEdit); // add edit to manager

   value = v; // finally, set the value
   notifyObservers();
}
```

Triggering Undo or Redo

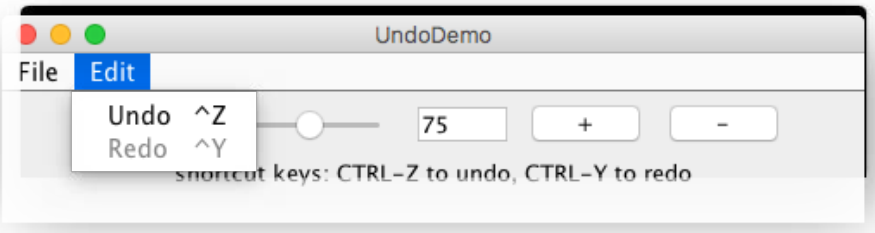
- Usually done with “undo” and “redo” menu items (with key Accelerators for CMD-Z, CMD-Y mapping)

```
public void undo() {
    if (undoManager.canUndo())
        undoManager.undo();
}

public void redo() {
    if (undoManager.canRedo())
        undoManager.redo();
}
```


Code Demo: UndoDemo

- Model handles all undo
  - `UndoManager` in Model
  - setters save `UndoableEdits` (uses closure)
  - methods added for undo state: `canRedo`, `canUndo`
- `MainMenuView` observes model to enable undo/redo menu items
- Menu has Accelerator keys (hotkeys)
- Note *the view doesn’t know anything about undo*, it just works



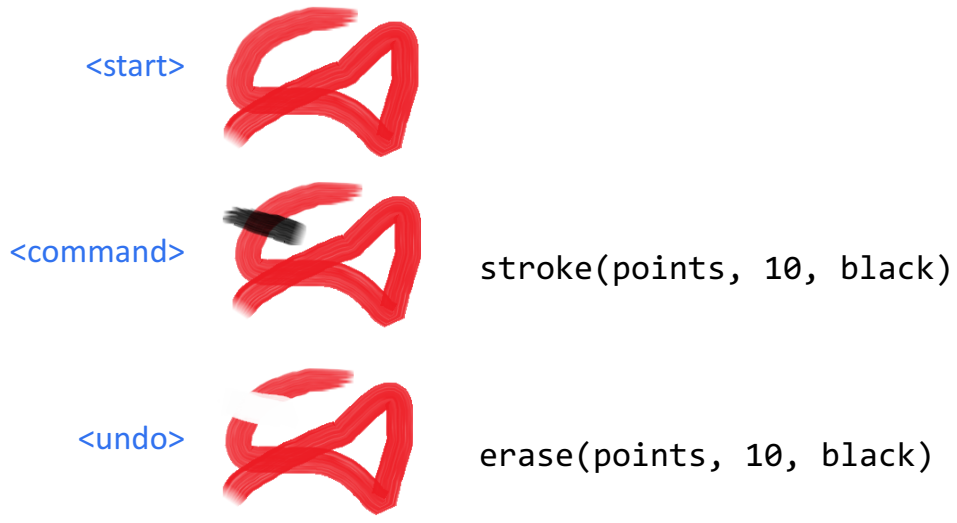
Java Undo Interfaces and Classes

- Interfaces
  - **`UndoableEdit`**: implemented by command objects. Key methods: `undo`, `redo`.
  - **`StateEditable`**: implemented by models that can save/restore their state. Key methods: `storeState`, `restoreState`
- Classes
  - **`AbstractUndoableEdit`**: convenience class for `UndoableEdit`
  - **`StateEdit`**: convenience class for `StateEditable`;
  - **`UndoManager`**: container for `UndoableEdit` objects (command pattern). Key methods: `addEdit`, `canUndo`, `canRedo`, `undo`, ...
  - **`CompoundEdit`**: "A concrete subclass of `AbstractUndoableEdit`, used to assemble little UndoableEdits into great big ones."

Command Undo Problems

- Consider a bitmap paint application

```
stroke(points, thickness, colour)
erase(points, thickness)
```



Solutions for “Destructive” Commands

- Option 1: Use forward command undo …
- Option 2: Use reverse command undo, but un-execute command stores previous state for “destructive” commands
  - that’s a Memento!
  - might require a lot of memory
  - why some applications limit the size of undo stack

Summary

- Benefits of undo/redo
  - enables exploratory learning
  - lets users recover from errors
  - lets users evaluate modifications (undo-redo cycle)
- Design
  - Undoable Actions: what can’t be / isn’t undone?
  - State Restoration: what part of UI is restored after undo?
  - Granularity: how much should be undone at a time?
  - Scope: is undo/redo global in scope, local?
- Implementation
  - Forward vs. Reverse undo
  - Command vs. Memento pattern