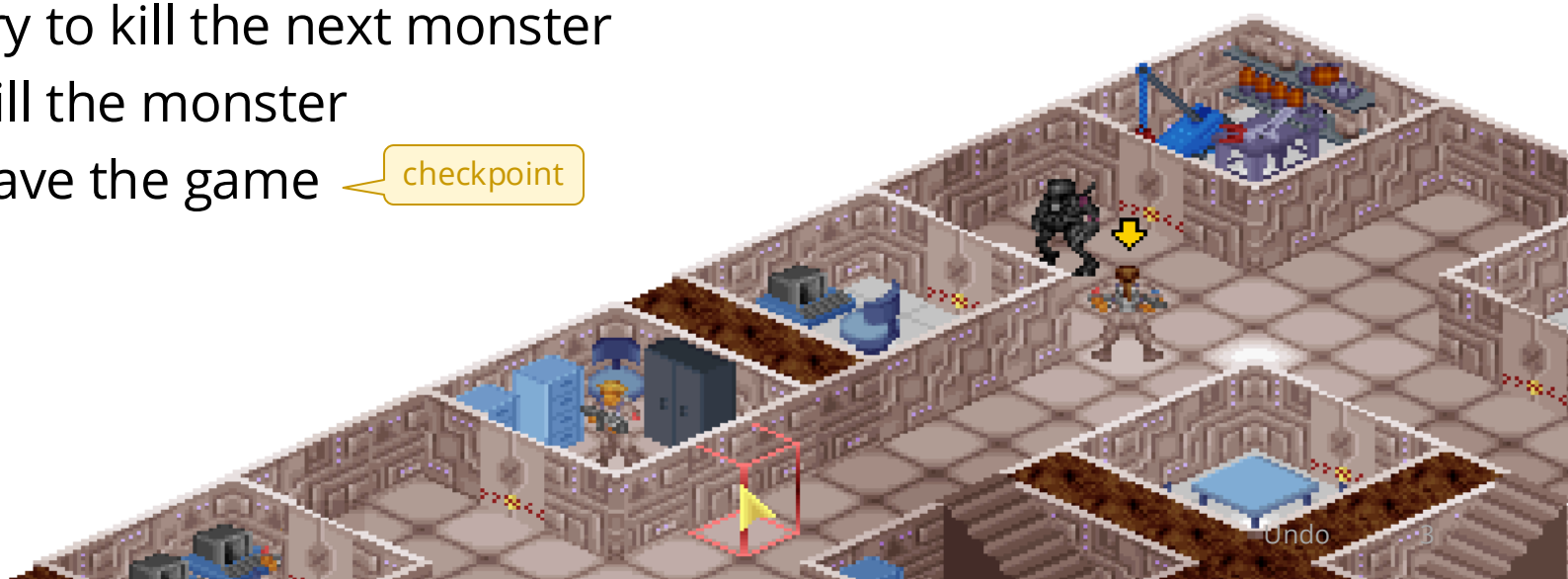
Undo*

- Principles, concepts
- Undo patterns
- Implementation

*Unless stated otherwise, “undo” means “undo/redo” in these slides.

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 (checkpoint)
  - You try to kill the next monster
  - You die
  - You reload the saved game ("undo")
  - You try to kill the next monster
  - You kill the monster
  - You save the game (checkpoint)

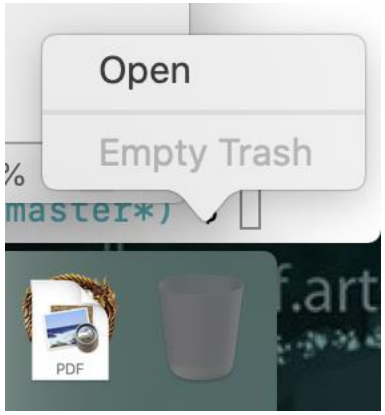


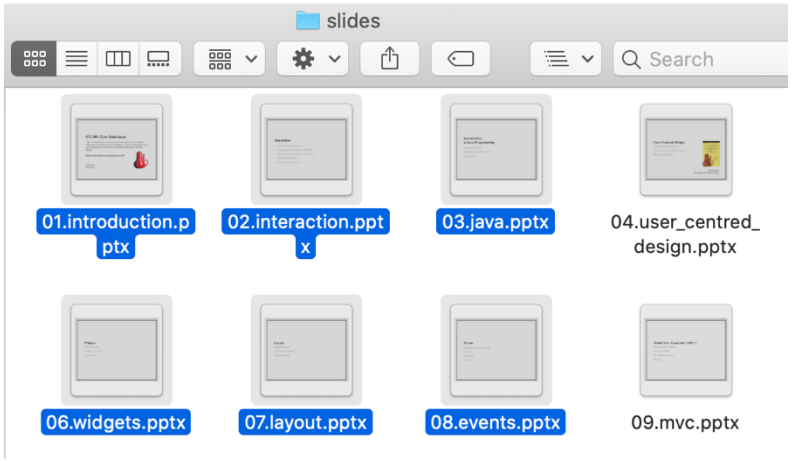
Undo Design Choices

1. **Undoable Actions**
2. **State restoration**
3. **Granularity**
4. **Scope**

Undoable Actions

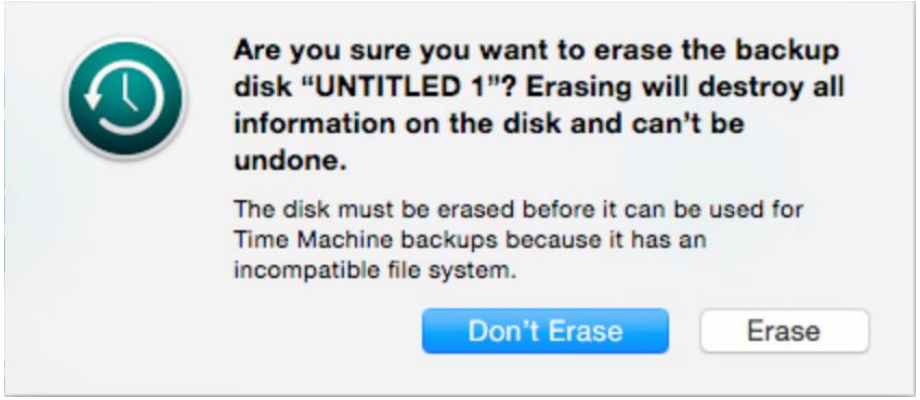
- Some actions may be omitted from undo
  -
- Some actions are destructive and not easily undone
  -
- Some actions can’t be undone
  -





Suggestions for Undoable Actions

1. All changes to "document" should be undoable
   - *All* such changes are in the Model
2. Changes to View (interface state) should be undoable ONLY if extremely tedious or require significant effort
   - Typically View changes are NOT undoable
3. Ask for confirmation before doing a destructive action which cannot easily be undone



State Restoration

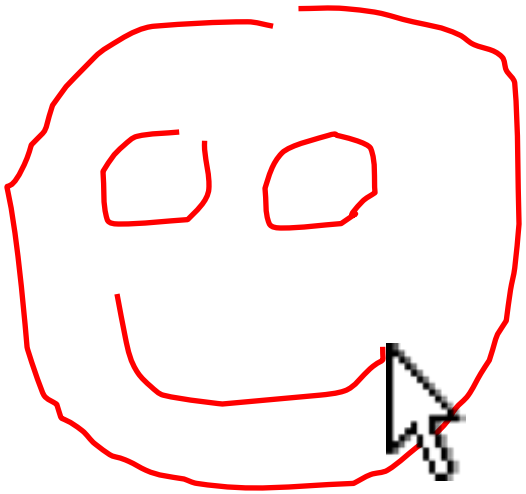
- What is the user interface state after an undo or redo?
  - e.g. highlight text, delete, undo … is text highlighted?
  - e.g. highlight text, delete, scroll, undo … scroll back to text?
- User interface state should be meaningful after redo action
  -
  -
  -

Granularity

- How much should be undone at one time?
- A **chunk** is conceptual change from one state to another
  - Interaction can be divided into undoable chunks
  - Undo reverses one chunk
- What defines one undoable “chunk”?

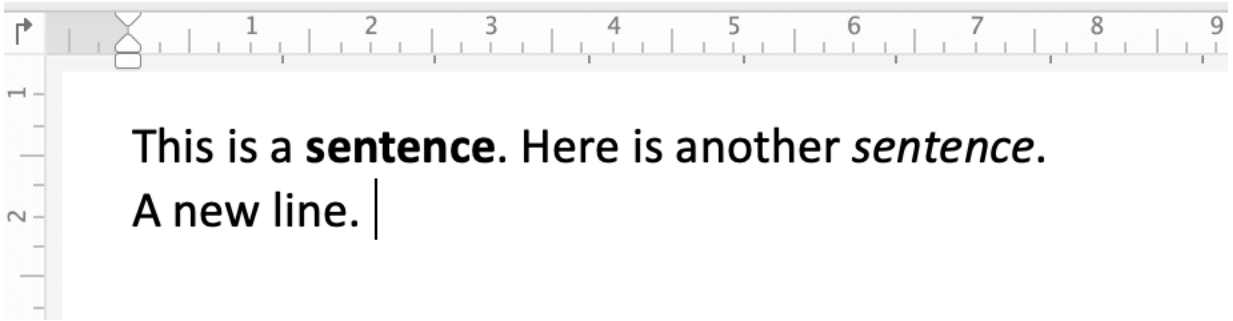
Granularity for Drawing Interactions

- What is a “chunk” to undo?



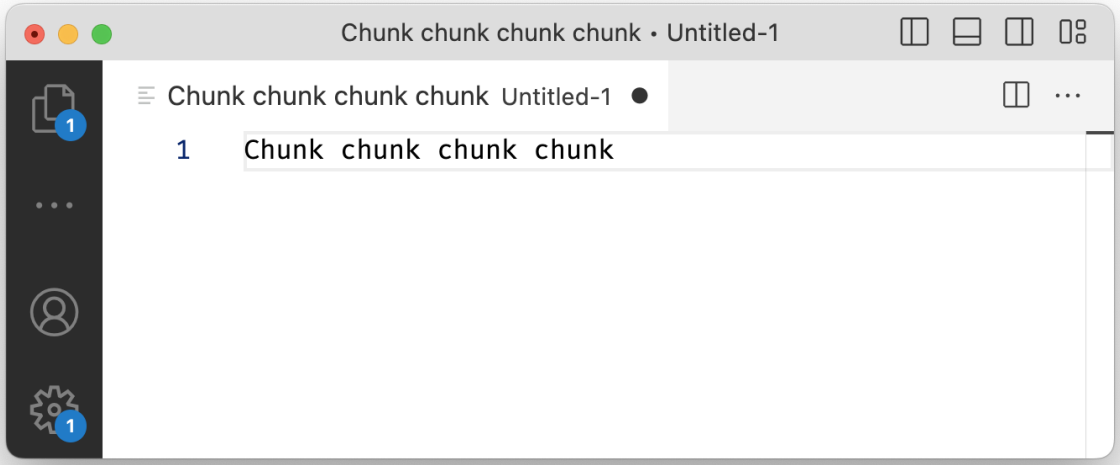
Granularity for Text Interactions

- What is a good undoable chunk for text?
  - Each typed character? (i.e. every keypress event)
  - Each syntactical unit? (i.e. a word, a sentence, etc.)
  - Between "commands"? (i.e. delimited by delete, bold, etc.)
  - Temporal sequence? (i.e. when edits separated by pause)
  - What about auto correct and suggestions? Are these chunks?



Granularity for Text Interactions

- Type some text, then press undo. What happens?
  - VS Code →
  - MS Word →
  - Google Docs →
  - Chrome search bar →
- Try typing sentences, changing formatting, deleting, ignoring or using suggestions, pausing or typing quickly, etc.



Implementing Undo

Two general approaches:

- **Forward Undo**
  - Start from base document, then maintain of list of changes to compute current document
  - Undo by removing last change from list when computing current document
- **Reverse Undo**
  - Apply change to update document, but also save "reverse" change
  - Undo by applying reverse change to document
- A **change record** defines a single transformation to the "document" (i.e. the state of the Model)

Forward Undo

- Save **baseline** document state at some past point:

  S^*

- Save **change records** to transform baseline document into current document state:

  $S = (c(b(a(S^*))$

- To undo last action, **don't apply last *change record:***

  $S' = \text{undo}(c(b(a(S^*)) = (b(a(S^*))$

Reverse Undo

- Save complete **current** document state:

  S

- Save ***reverse*** **change records** to return to previous state:

  $\{c^{-1}, b^{-1}, a^{-1}\}$

- To undo last action, **apply last reverse *change record:***

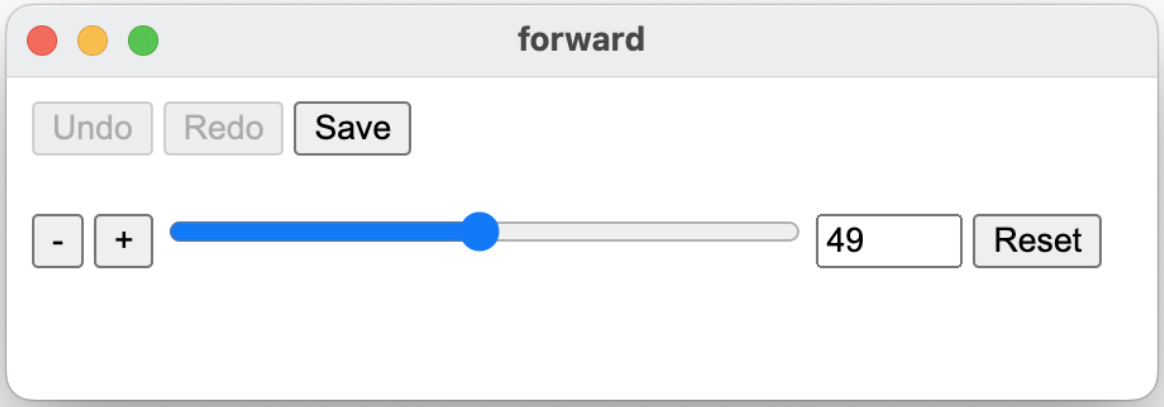
  $S' = \text{undo}(S) = c^{-1}(S)$

Implementation with Stacks

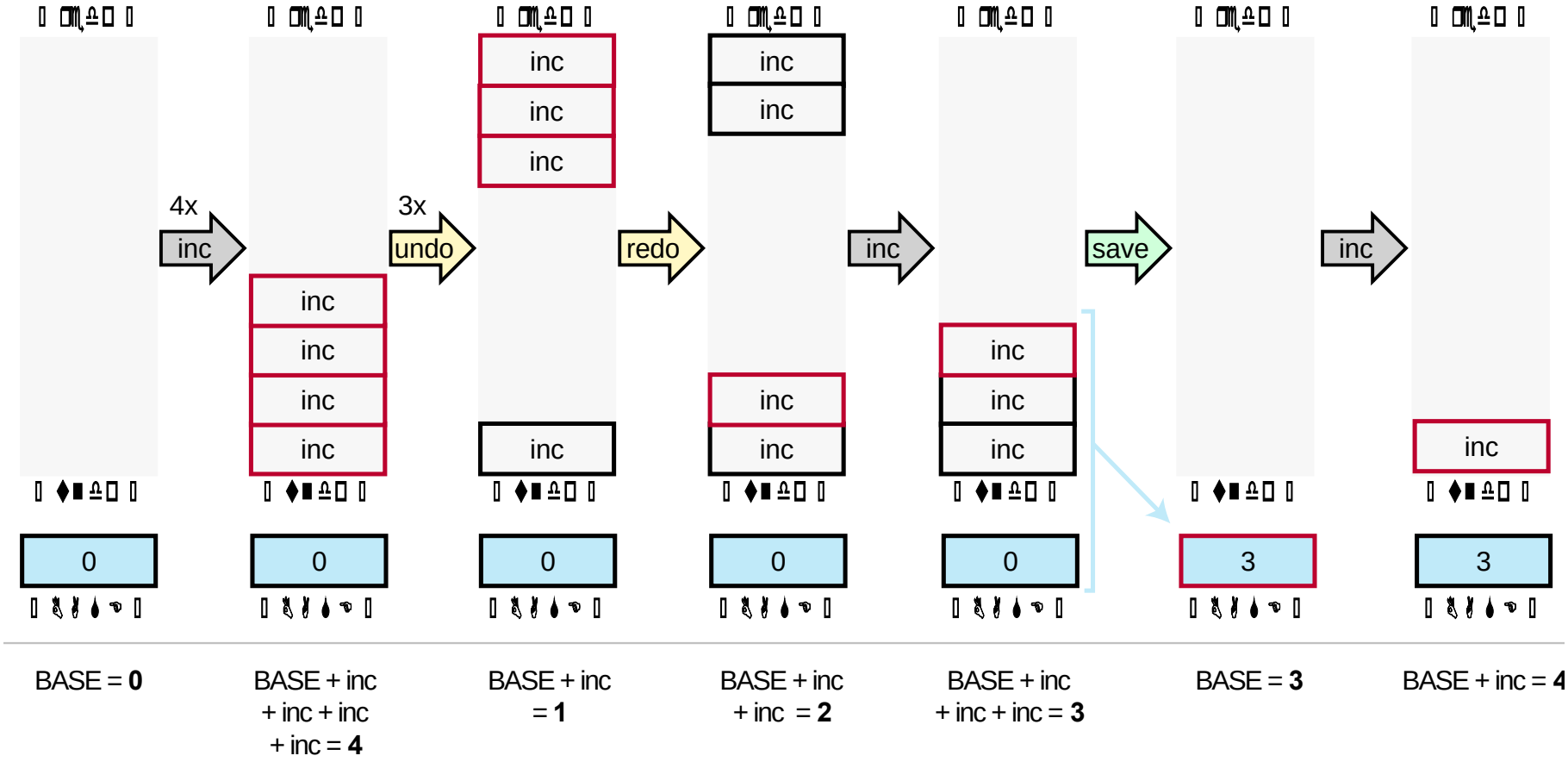
- Using either of these options requires two stacks
  - **Undo stack**: all change records, saved as you perform actions
  - **Redo stack**: change records that have been "undone" (needed to reapply them with redo)

forward

- A simple counting app with undo/redo



Forward Undo



Forward Undo Command

In undo.ts, define interface for forward command:

```
export interface Command<State> {
  do(state: State): State;
}
```

Defining a generic type

In model.ts, create a specific command for each action, e.g.

```
increment() {
  // add command to undo stack
  this.undoManager.execute({
    do: (state) => state + 1,
  } as Command<number>);
  ...
}
```

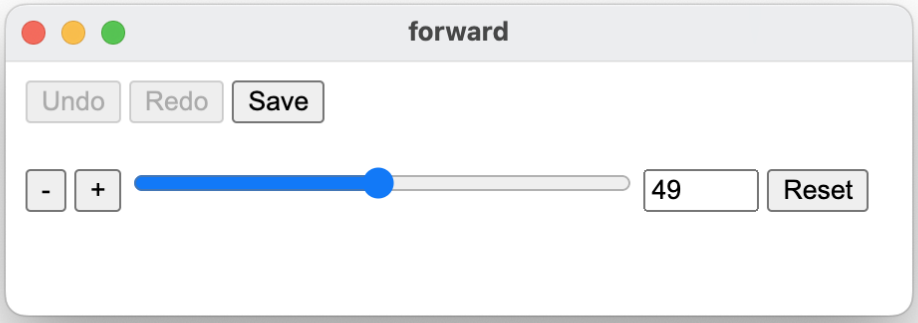
Creating specific instance of generic type

Learn TypeScript Generics In 13 Minutes

- https://youtu.be/EcCTIExsqml

forward

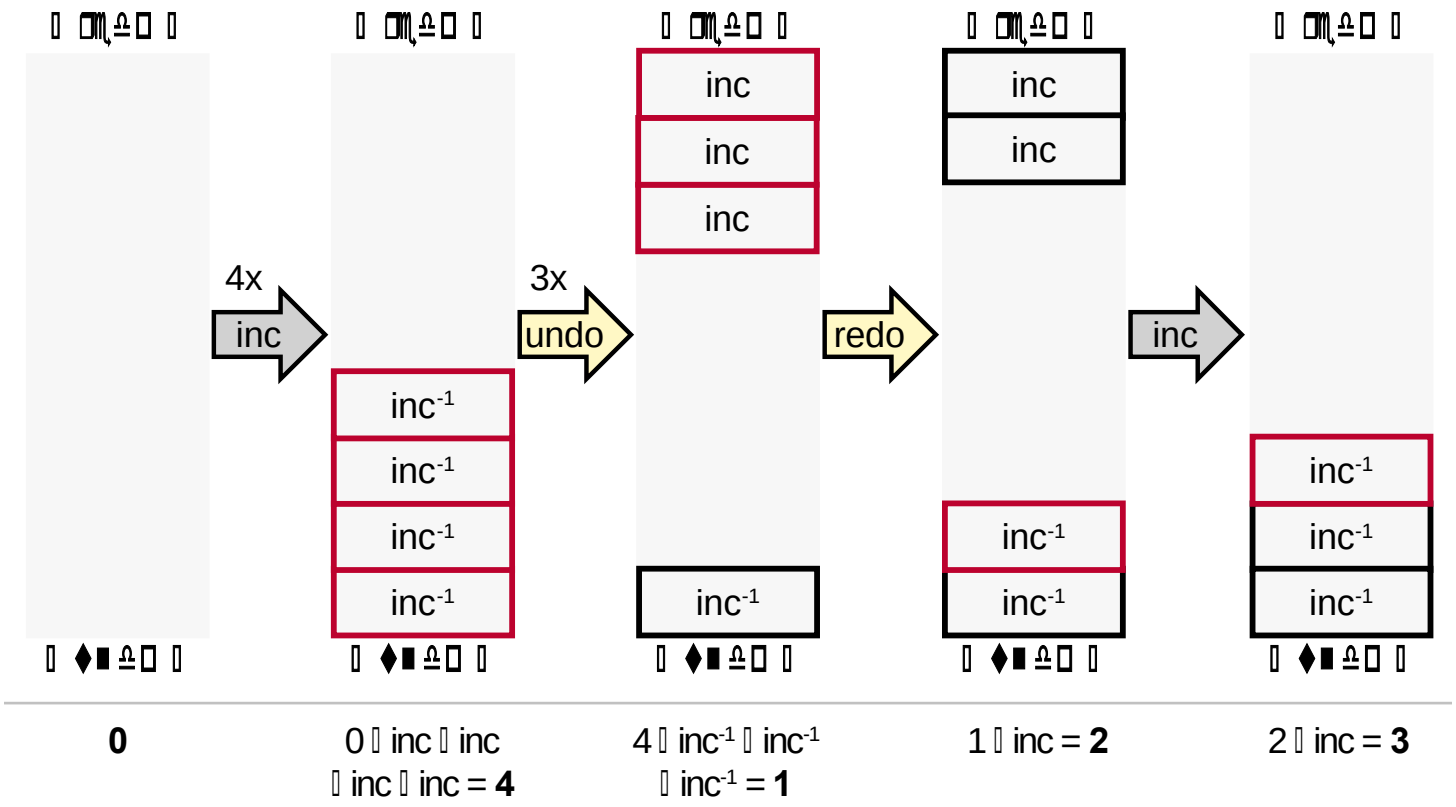
- `Command` interface
  - has single "do" command with a Generic Type
- `UndoManager` class
  - two stacks: undo, redo
  - `execute` to add command to "undo" stack
  - `computeState` called by Model
  - `undo` moves command from undo to redo stack
  - `redo` moves command from redo to undo stack
- `Model` class
  - set `count`, `increment`, `decrement`, reset all create "do" command and send it to `UndoManager`
  - get count asks `UndoManager` to `computeState`
  - save resets the `baseState`



Reverse Undo Command Pattern

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

Reverse Undo



Command in "undo.ts"

Command interface

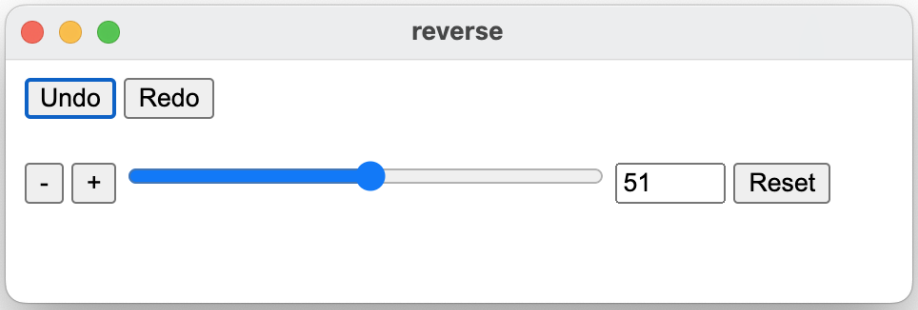
```
export interface Command {
  do(): void;
  undo(): void;
}
```

Example Command for increment

```
{
  do: () => {
    this._count++;
  },
  undo: () => {
    this._count--;
  },
} as Command
```

reverse

- UndoManager is simpler than forward undo
  - redo executes the command (do)
  - undo executes the reverse command (undo)
- Model
  - each mutation method creates a do/undo command
  - undo and redo methods notify observers
- Demo
  - example of poor granularity using "input" event (instead of "change")



Reverse (undo-chunking.ts)

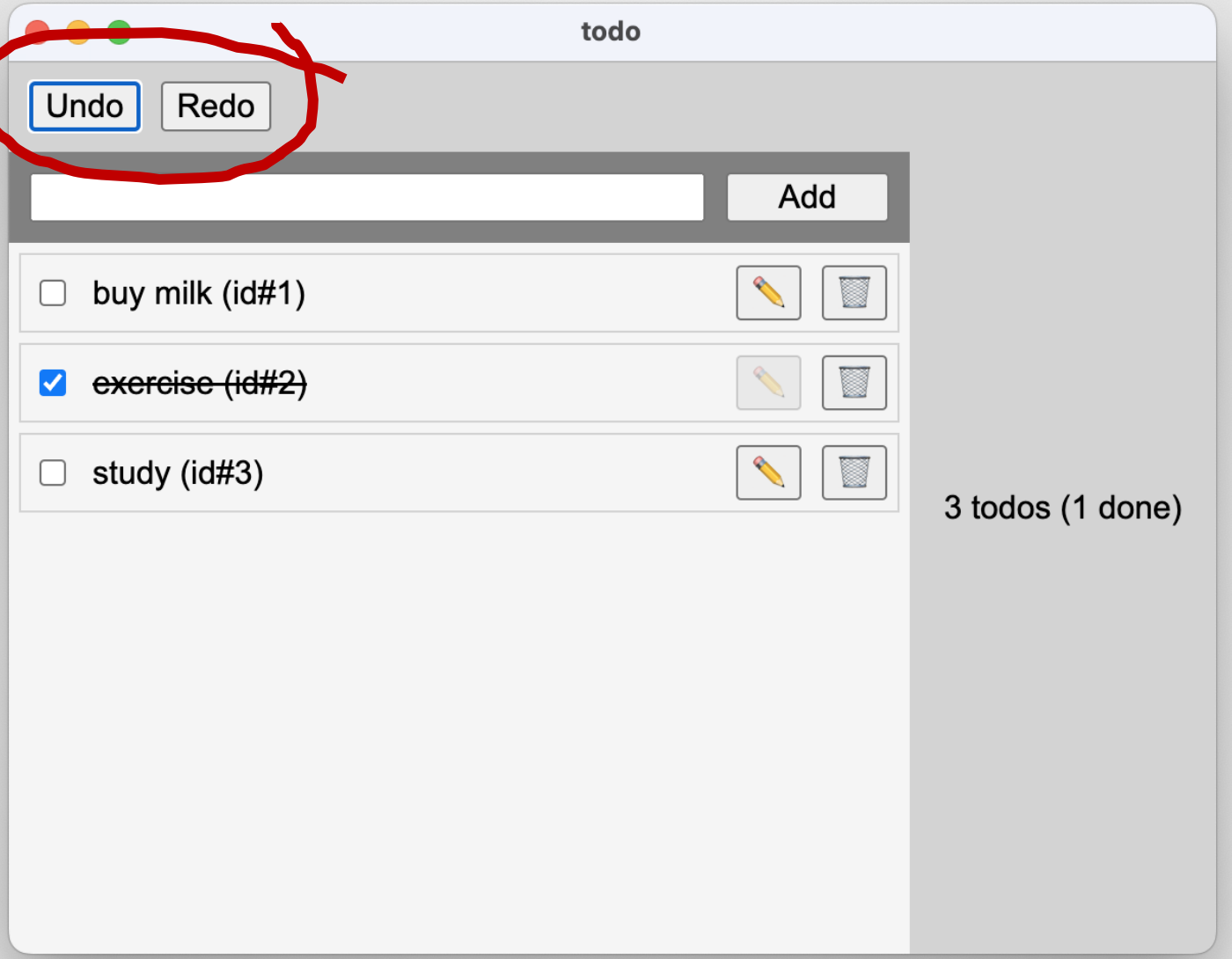
- Switch undo to undo-chunking in Model
  - Undo now uses timeout to chunk consecutive actions
  - Stores sequence of commands in a chunk
  - Undo and redo work on chunks, not individual commands
- Demo
  - Use “input” event for range to show how it works
  - Set the timeout to 1000ms to more easily see it working
  - Numeric up/down also good example

Example Text Editor Commands and Reverse Commands

- Available Commands:

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

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



Todo demo from HTML CSS lecture **with undo**

todo

only the `Model` changed to support undo/redo

- Added a View with Undo and Redo buttons, *but all other Views remained exactly the same as the HTML CSS demo*
- `UndoManager` and standard undo methods added to Model
  - `undo`, `redo`, `canUndo`, `canRedo` (just like prev demos)
- do and undo commands added to `Model` methods that mutate data

Reverse Change Record Implementation Options

- Option 1: **Command** pattern
  - save command and "reverse command" to change state
- Option 2: **Memento** pattern
  - save snapshots of each document state
  - could be complete state or difference from “last” state

memento

- Simple example of undo using "mementos"
- Executing undo moves top memento to redo stack, then uses new top of undo stack to set the Model state
- Needs a base memento (set in constructor)
  - When undo stack is empty, base memento is used
- Implementation uses TypeScript generics
  - `UndoManager` and `Memento` work with any type of Model state

```
interface Memento<State> { state: State; }
```

generic type

  - Demo uses:

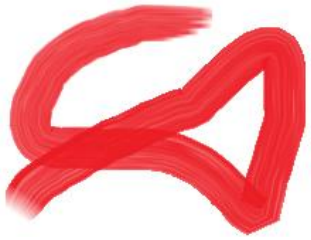
```
Memento<number>
```

Reverse Command Undo Problems

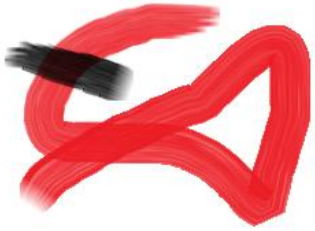
- Consider a bitmap paint application

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

<start>



<command>



```
stroke(points, 10, black)
```

<undo>

```
erase(points, 10, black)
```

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