CSE 1325: Object-Oriented Programming
Lecture 10

User Interfaces:
History and Writing Your Own

Easy as Pie!

Mr. George F. Rice
george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

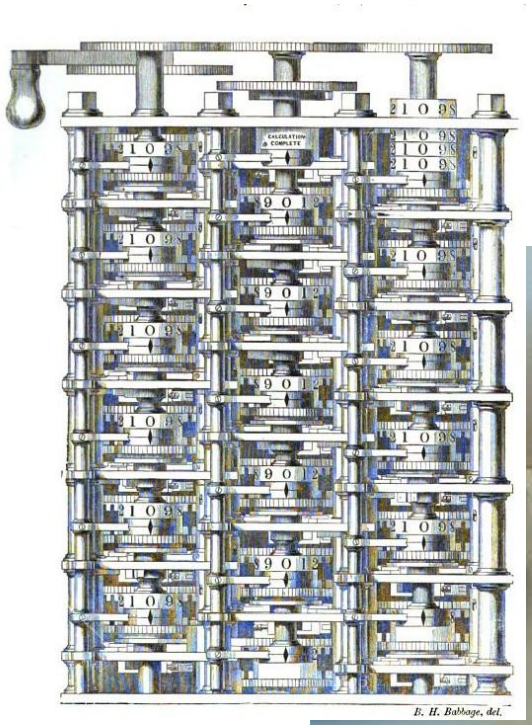
Piece of Cake!

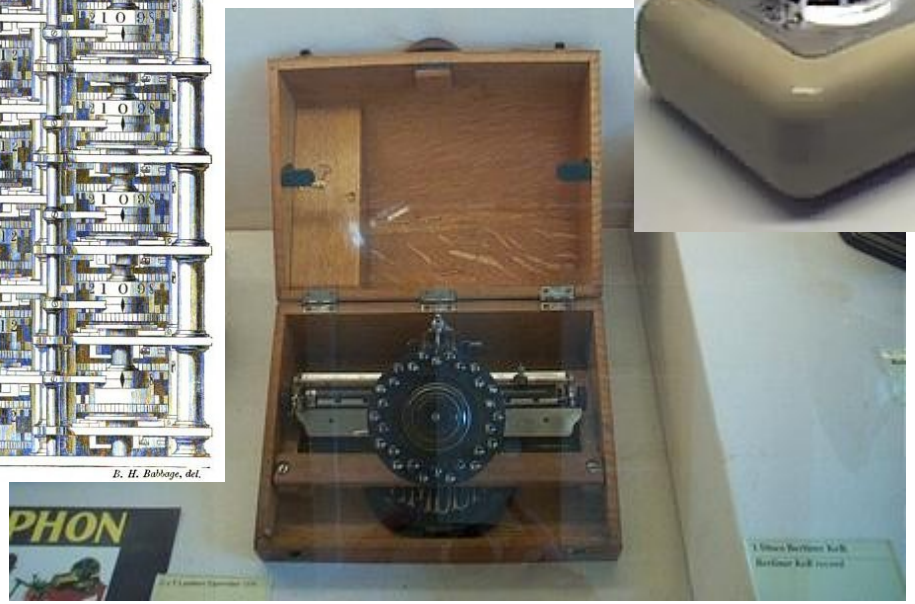
Don't cry because it's over; smile because it happened.

Pie by congerdesign and Cake by SZimmermann_DE per the Pixabay License

Question

- When was the first user input device deployed for automated computing machines?





Babbage engine by Woodcut via a drawing by Benjamin Herschel Babbage - Google books, Public Domain, https://commons.wikimedia.org/w/index.php?curid=25778944
Rotary keyboard by Kaihsu Tai - Uploaded and copyright Kaihsu Tai, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=95724
Joystick by Solkoll - photo Solkoll, Public Domain, https://commons.wikimedia.org/w/index.php?curid=617869
Teletype image: Public domain, published in the US between 1923 and 1977 without a copyright notice. https://commons.wikimedia.org/wiki/File:What-is-teletype.jpg

Paper Tape

- Early user interfaces were based on paper tape...
  - Paper tape (with 5, 6, 7, or 24 holes per row) date back to **1725** for automating weaving looms and (in 1876) the “player piano”
  - First used in 1846 to prepare telegrams for transmission
  - Punch machine separate from reader
  - Excellent confetti generator!

Photo by Bad Germ CC-BY-SA-3.0 Unported

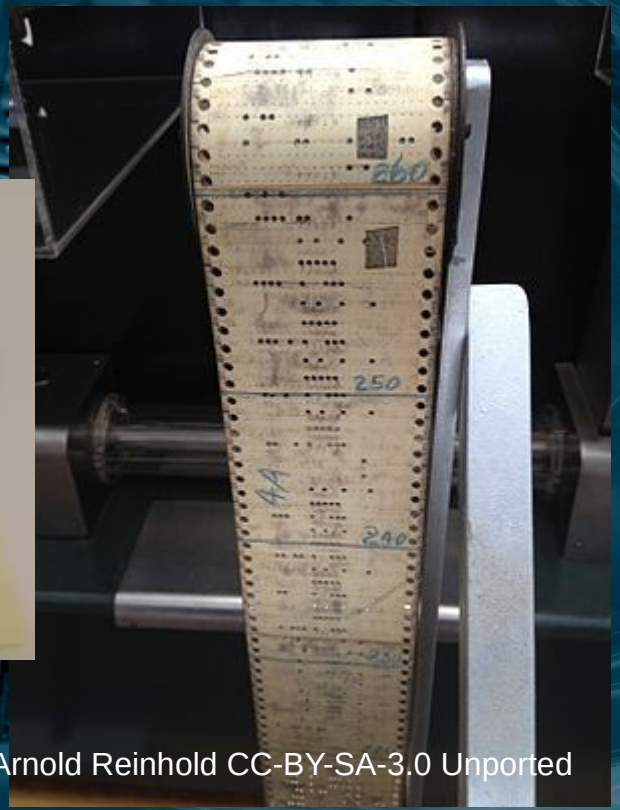
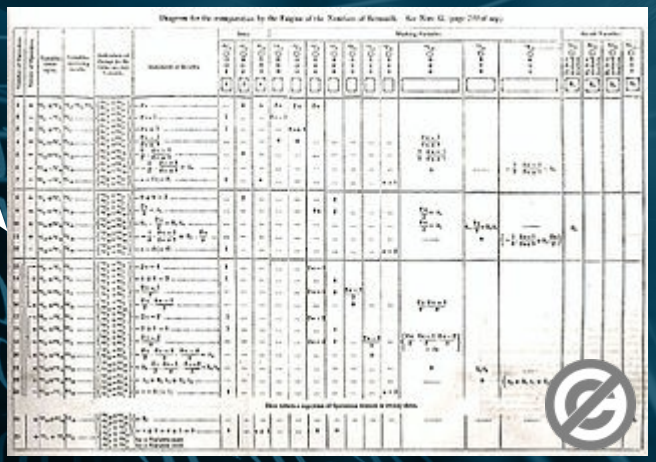
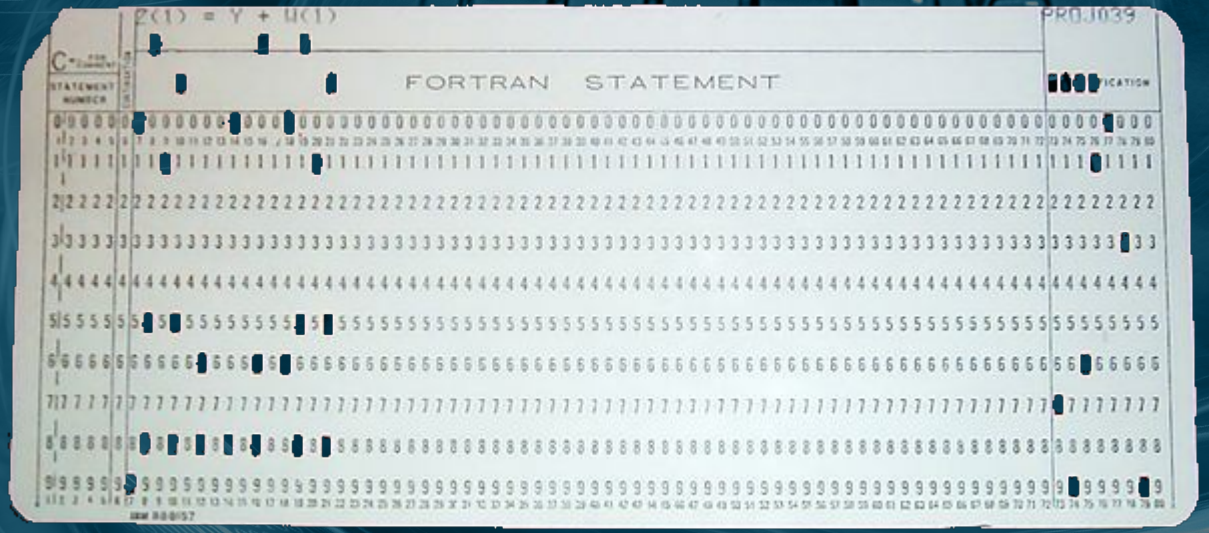


Photo by Arnold Reinhold CC-BY-SA-3.0 Unported

Punch (Hollerith) Cards

- … and punch cards aka Hollerith cards
  - Cards with 80 columns, often driving looms and musical instruments, starting in **1832**
  - Charles Babbage proposed use of cards for the Circulating Engine Store in his Analytic Engine design of 1834
    - Lady Ada Lovelace's algorithm to produce Bernoulli numbers was the first computer program... designed for cards
  - Cards were used to calculate the 1890 census

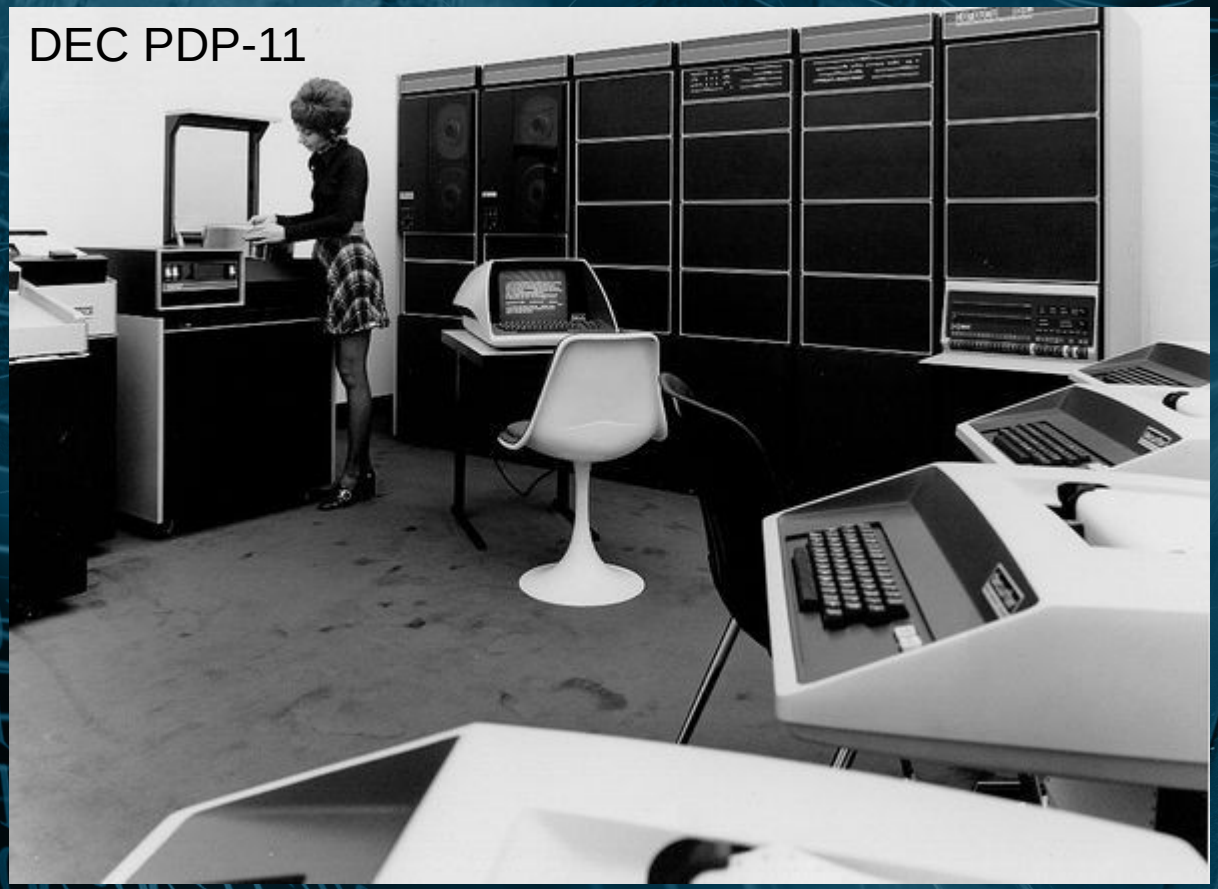




Fortran code: photo by Arnold Reinhold CC-BY-SA-2.5 Generic

User Input from Persistent Memory

Disk Packs 1965



Fair Use

8” 1971 80K **5¼” 1976 360K**

Public Domain

3½” 1986 1.44M

https://en.wikipedia.org/wiki/Floppy_disk#/media/File:Floppy_disk_2009_G1.jpg

CF 1994 8M

Fair Use

Interactive Switches and Tubes

- Digital computers supported physical switches and lights or Nixie tubes on the front panel
  - To boot the computer, just load the program via switches
  - Debug by watching light patterns

PDP-8 photo by Florian Schäffer CC-BY-SA-4.0

Georg-Johann Lay Gnu FDL 1.2+

PDP = Programmable Data Processor

Keyboards and Printers / CRTs

- Keyboards and thermal printers or displays brought the Command Line Interface (CLI) to reality around 1965

Dave Fischer CC-BY-SA-3.0

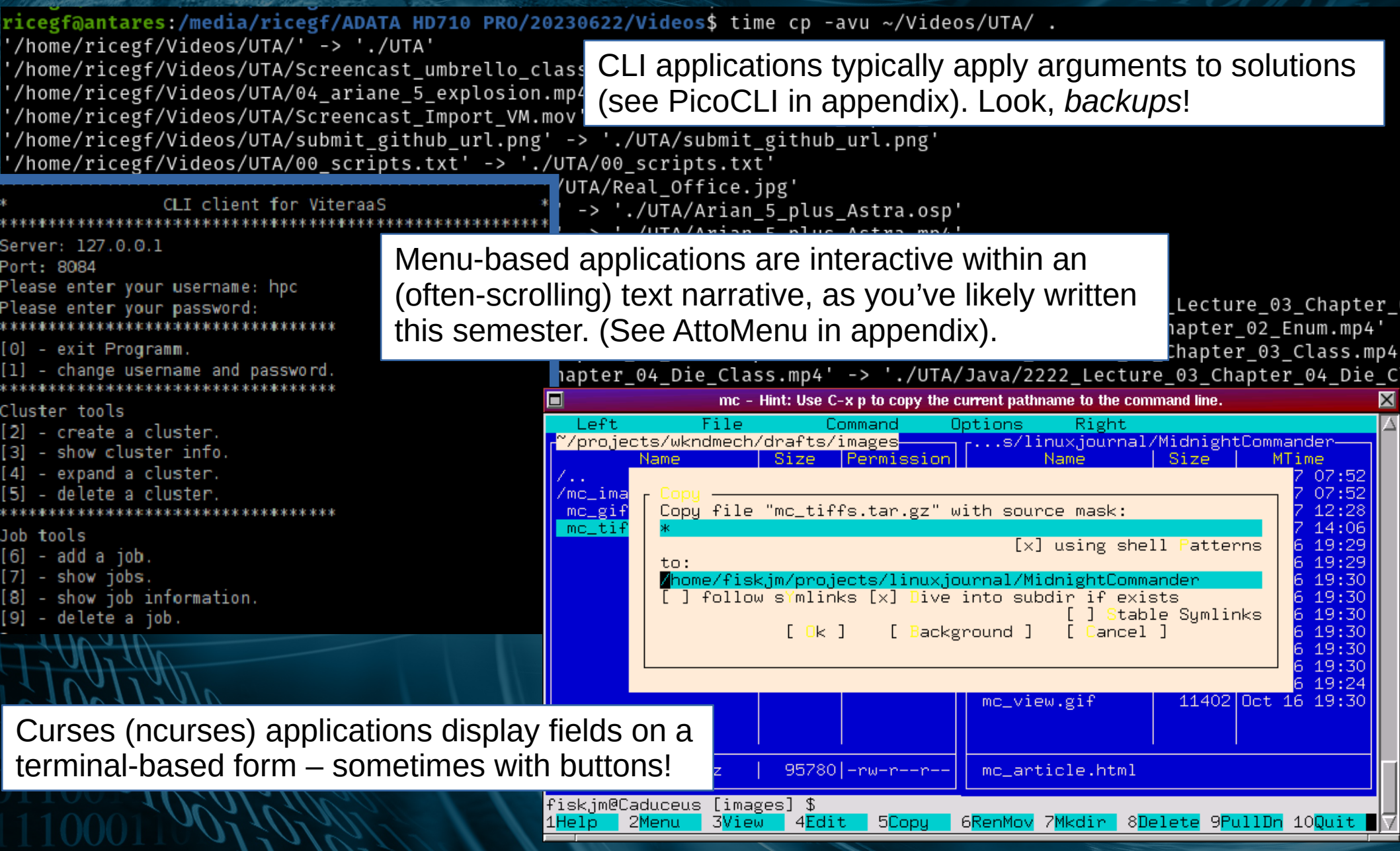
Jason Scott CC-BY-SA-2.0 Generic

Command Line Interfaces

CLI applications typically apply arguments to solutions (see PicoCLI in appendix). Look, *backups*!

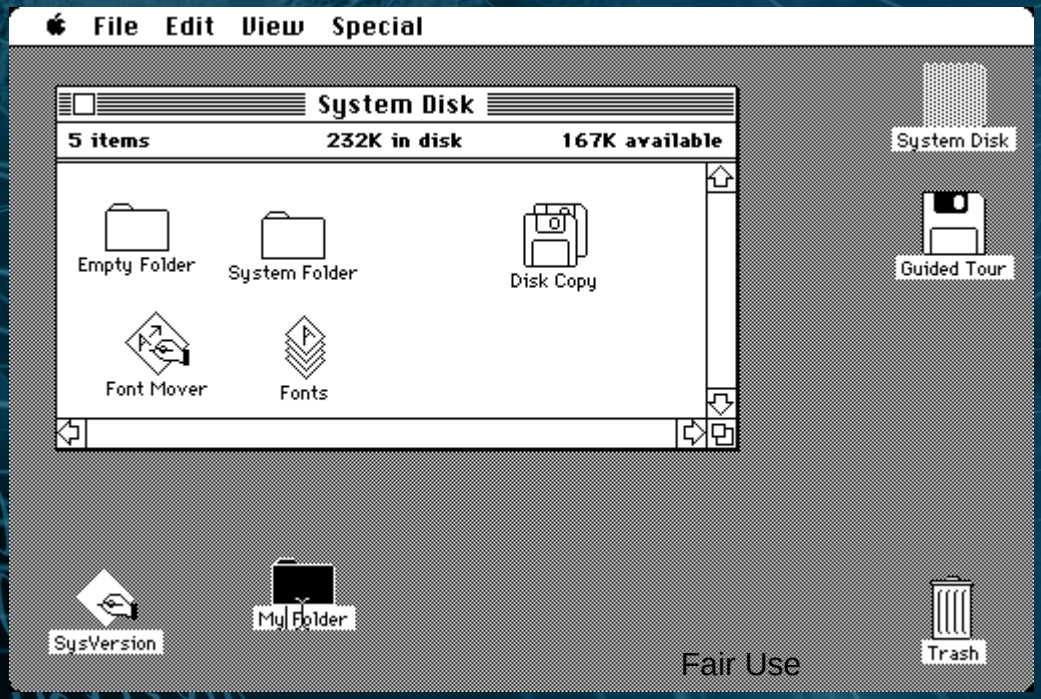
Menu-based applications are interactive within an (often-scrolling) text narrative, as you've likely written this semester. (See AttoMenu in appendix).

Curses (ncurses) applications display fields on a terminal-based form – sometimes with buttons!



Graphical User Interfaces

- In 1984 began the rise of the Graphical User Interface
  (Windows Icon Mouse Pointer, or WIMP)

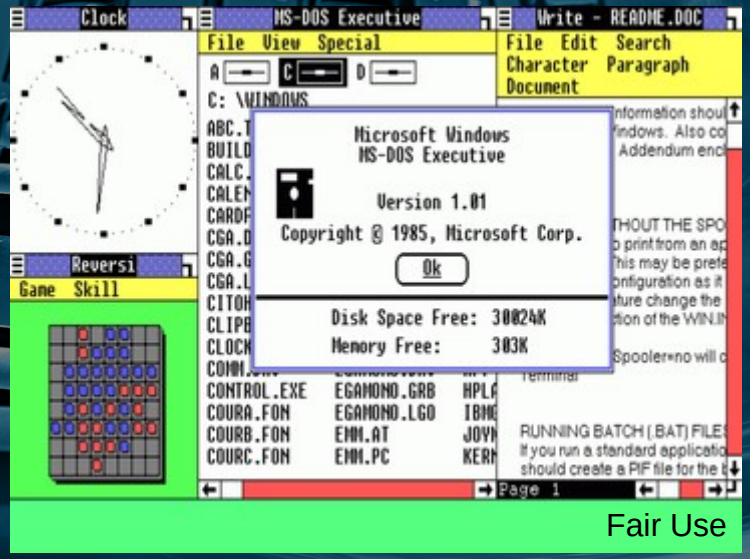


Fair Use

Apple Macintosh OS 1.0 Desktop

Fair Use

Xerox Star Compound Document



Microsoft Windows 1.01 →

Fair Use

Modern Graphical User Interfaces

- GUIs matured over time into “push button software” with rich mouse interaction

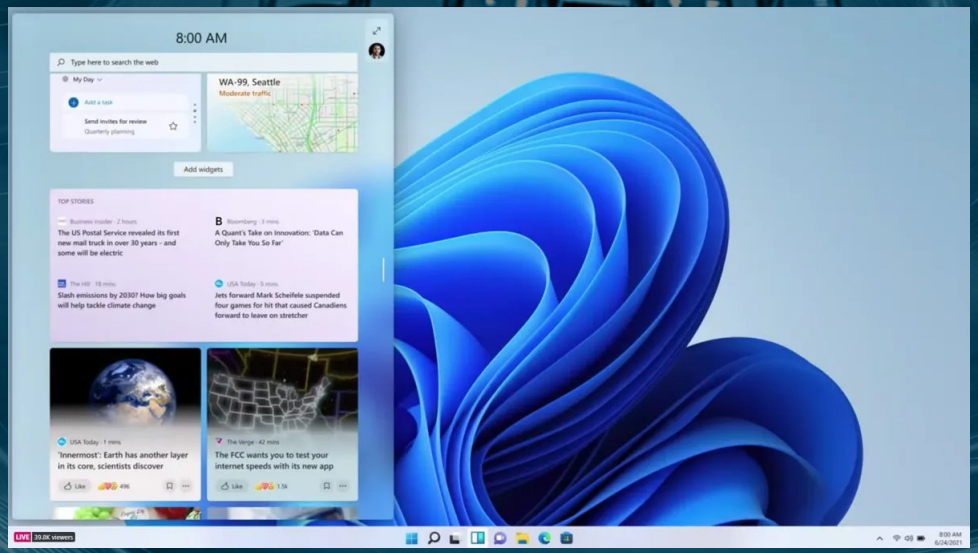
Linux Mint 21 Desktop

Fair Use

Apple Macintosh OS X 10.14 Desktop

Fair Use

Microsoft Windows 11 →



Fair Use

Touch / Gesture User Interfaces

- In 2006, the Touch Interface...
  (also called the Gesture Interface)

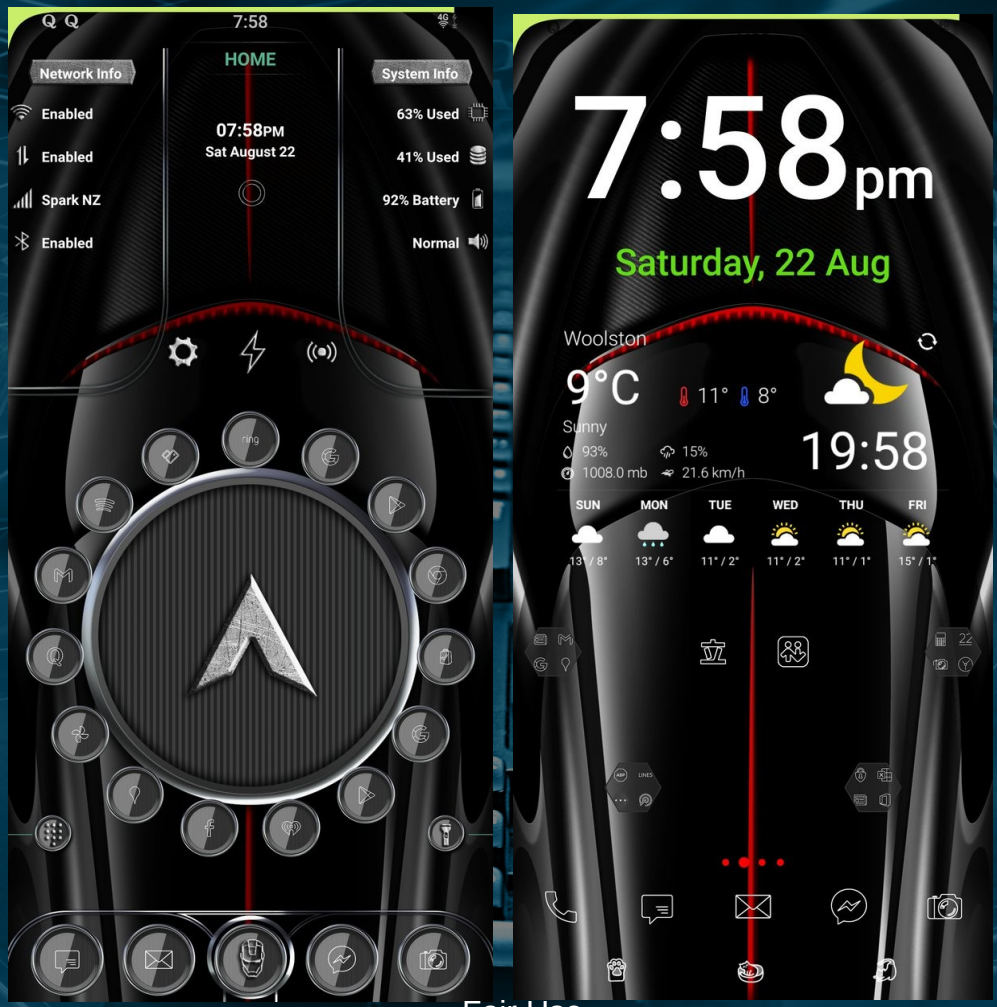
Fair Use

Modern Touch User Interfaces

- ...which also evolved and is now configurable

Fair Use

Apple iOS 16 Lock & Home Screens



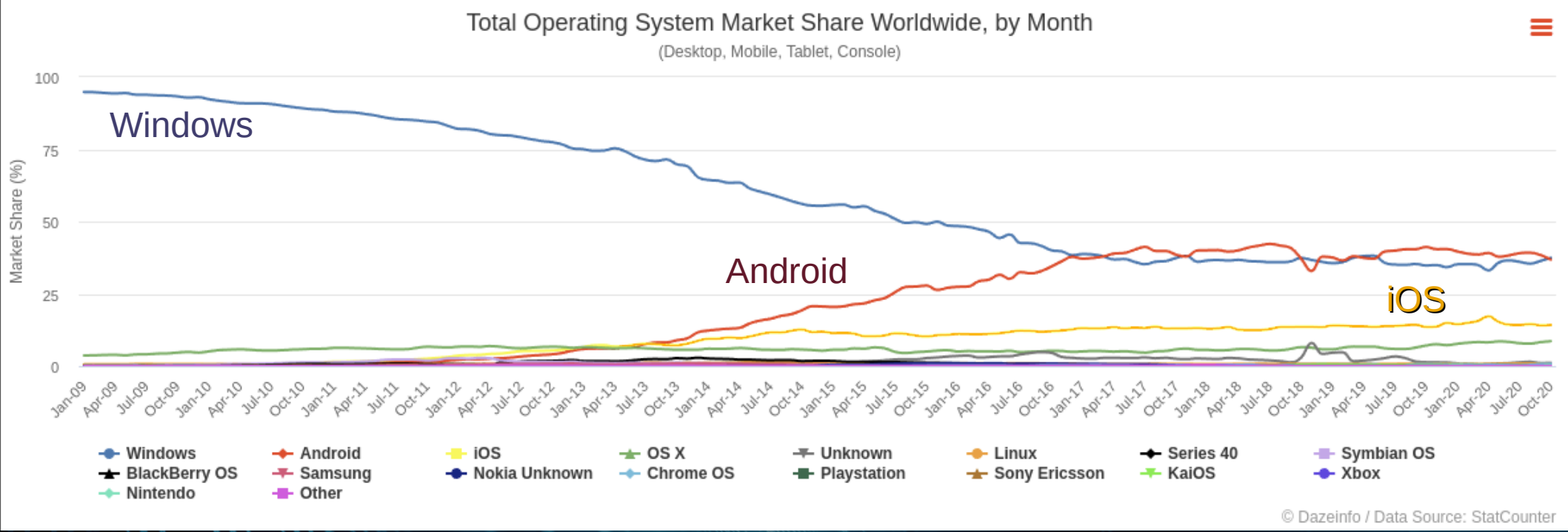
Fair Use

Android 12 Screenshots

Touch User Interfaces

- Touch has overtaken the desktop worldwide in total sales
  - But businesses & software developers still rely on desktops

The below graph represents the world's most popular operating systems by market share, starting from January 2009 to the recently completed month. Windows, Android, iOS, OS X and Linux are currently dominating the global OS market, across all platforms.



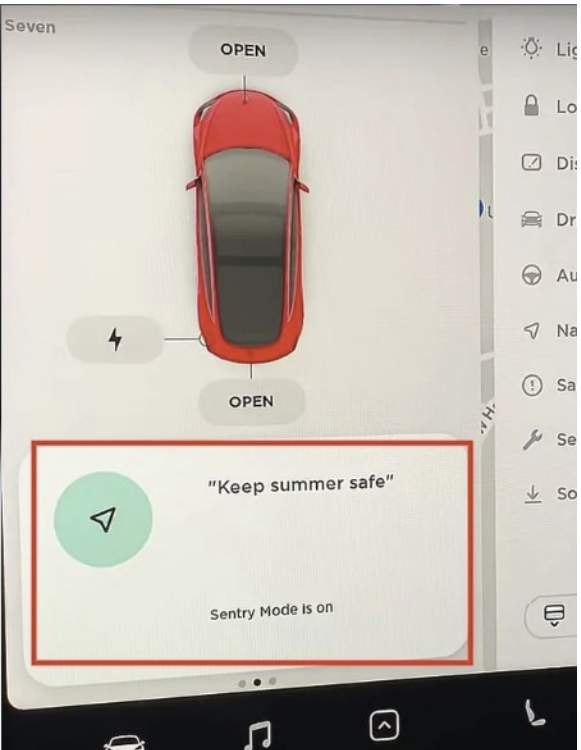
Total OS Worldwide Market Share from DazeInfo, via StatCounter. Fair use for educational purposes is asserted.
https://dazeinfo.com/2020/05/05/total-operating-system-market-share-worldwide-by-month-graphfarm/

Voice User Interfaces

In 1996, Phillips introduced Voice Dial™, an early Voice User Interface (VUI)

In 2011, Apple announced the Siri general VUI

In 2014, Amazon introduced the Alexa Echo family of VUIs



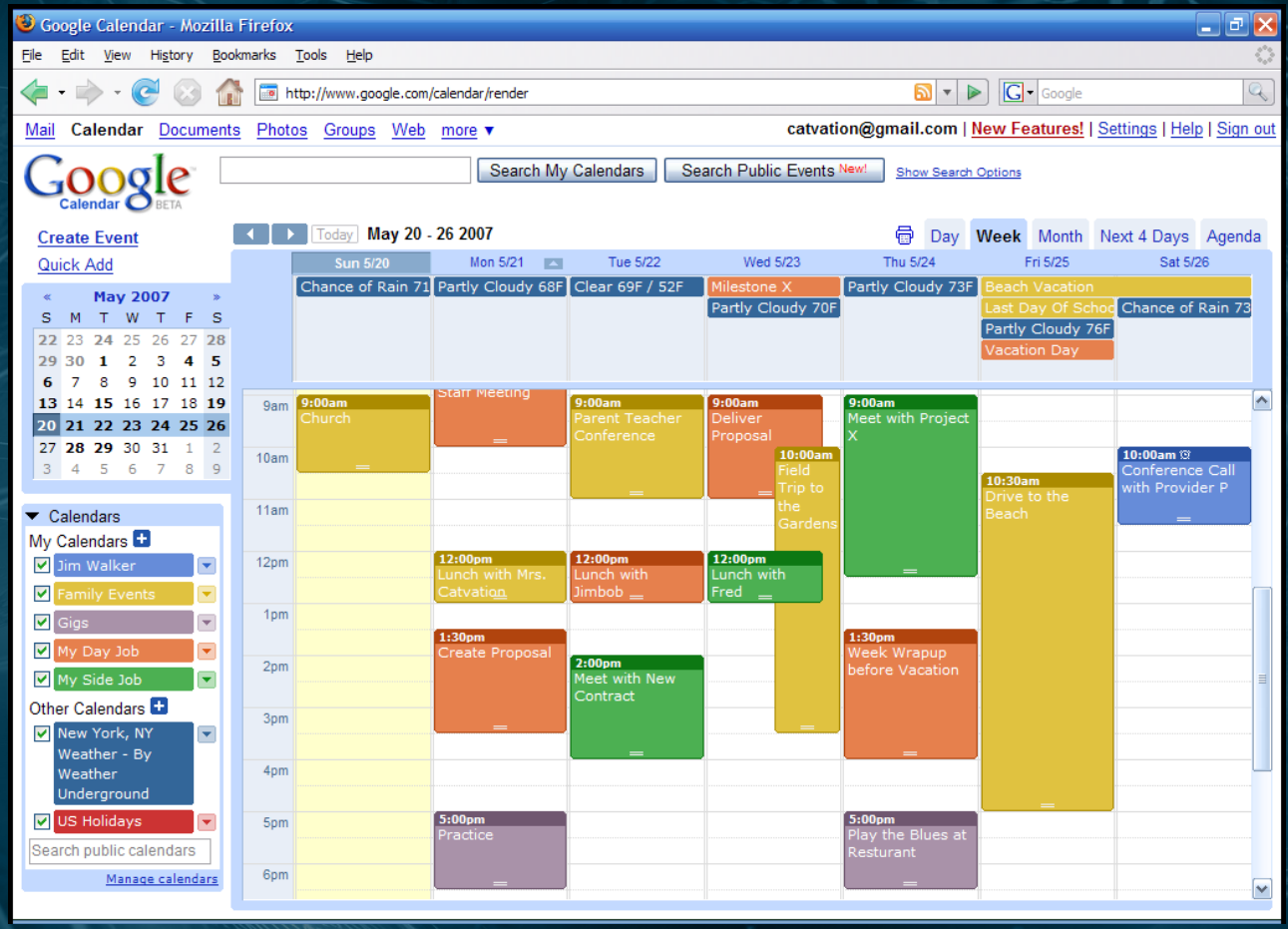
Drivers may benefit greatly using voice commands while driving (say) a Tesla vehicle

Seeking the Ultimate User Interface

http://xkcd.com/1367/

Modern Web User Interfaces

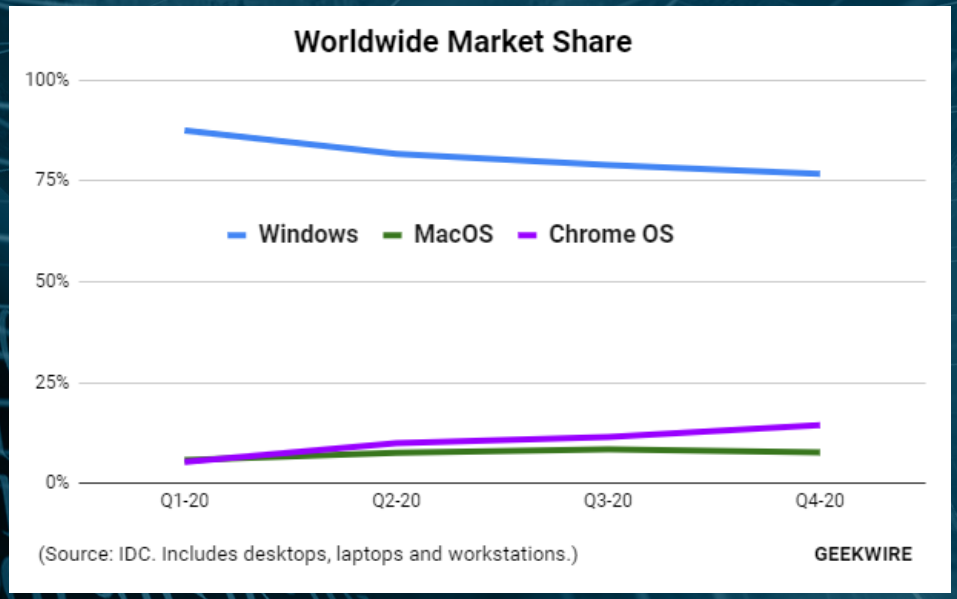
- In 1995, with the introduction of JavaScript, web applications began to evolve... slowly
- XMLHttpRequest (1999)
- AJAX (2005)
- HTML 5 and Chromebooks (2011)



Google Calendar on Firefox Browser

Modern Web User Interfaces

- Web UI devices in the US (not world) now outsell Macs and dominate education due primarily (IMHO) to
  - Simple support / updates
  - Android app support (2016)
  - Native Linux apps (2018)



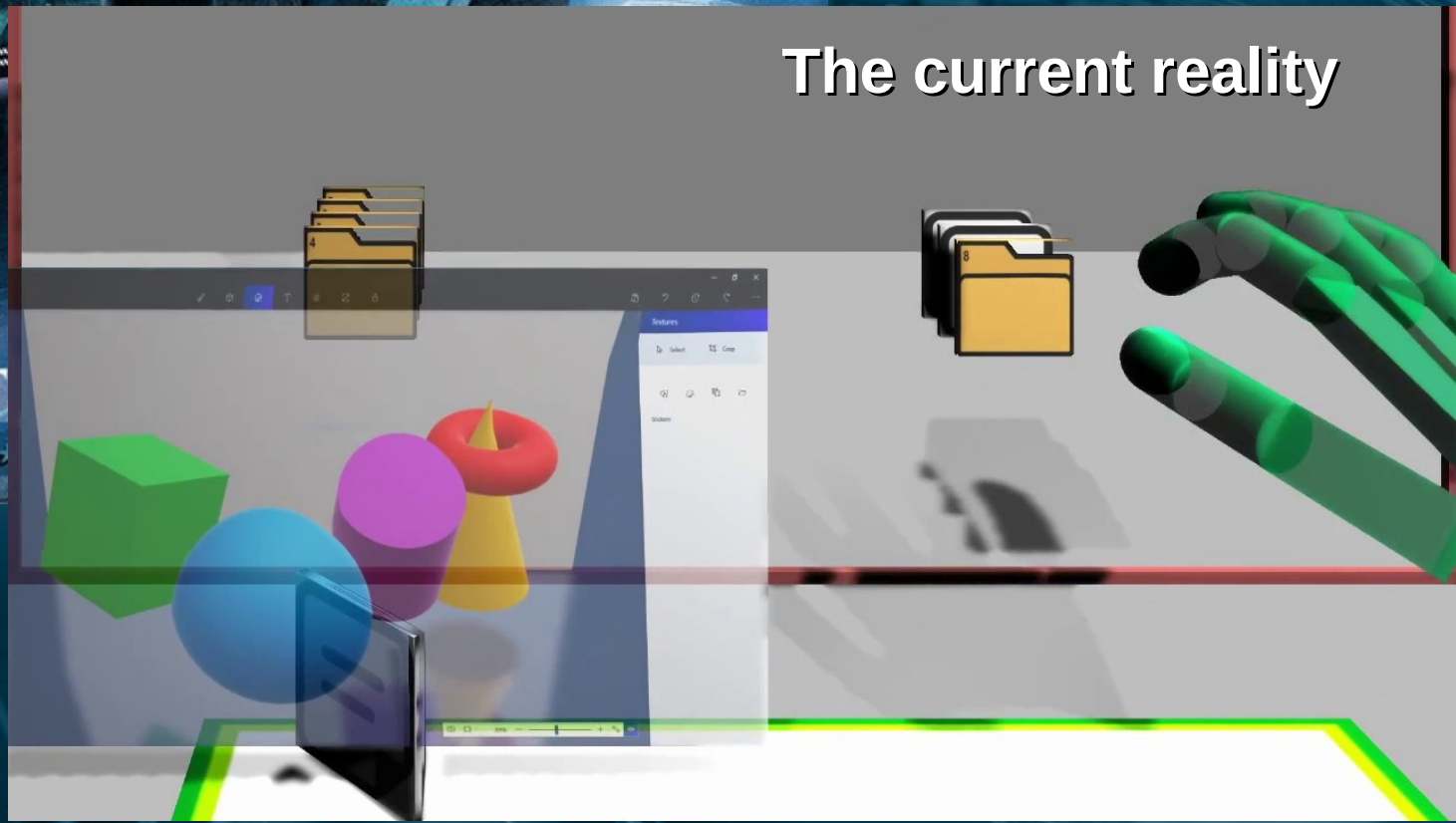
K-12 Mobile Computing Shipments

Worldwide Market Share from GeekWire, February 16, 2021. Educational fair use is asserted.
https://www.geekwire.com/2021/chromebooks-outsold-macs-worldwide-2020-cutting-windows-market-share/
K-12 Mobile Computing Shipments copyright 2018 ZDNet. Educational fair use is asserted.
https://www.zdnet.com/article/windows-pcs-gain-share-in-k-12-in-the-u-s-but-chromebooks-still-dominate/

Virtual Reality UI?

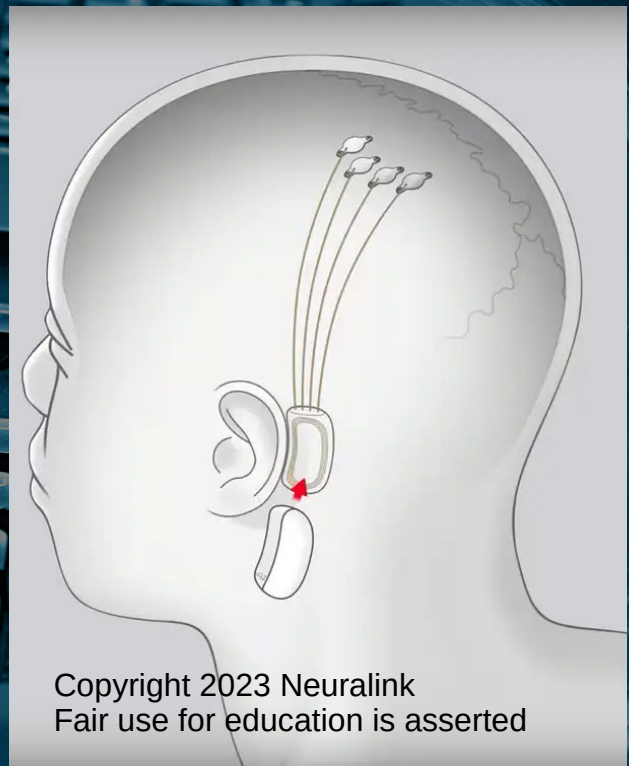
The dream

The current reality



Direct Brain-CPU Interface

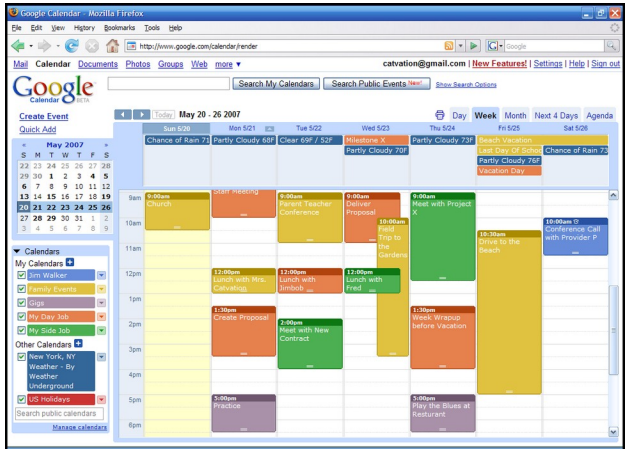
- The CPU could monitor brainwaves as well as stimulate the brain
  - Quadriplegic Internet access or prosthetic control
  - Avatar-like robotic control in hazardous areas
  - Seizure or stroke rapid response
  - Access to expanded sensors
    - Geordi La Forge's visor
    - "See" temperature, RF energy, wind/gases, microorganisms, magnetic fields, tele/microscope
    - Secure brain-to-brain communication
      - Shared thoughts and experiences



Most Common User Interfaces
(with some popular language options)

Mouse-driven desktop / laptop
C# (Windows), Swift (Mac),
C / C++, Python, Java Swing (Linux)

Voice
Python / js / Node



Web
js / HTML5 / CSS / Python
Server-side Java

Touch
Java (Android)
Swift (iOS)

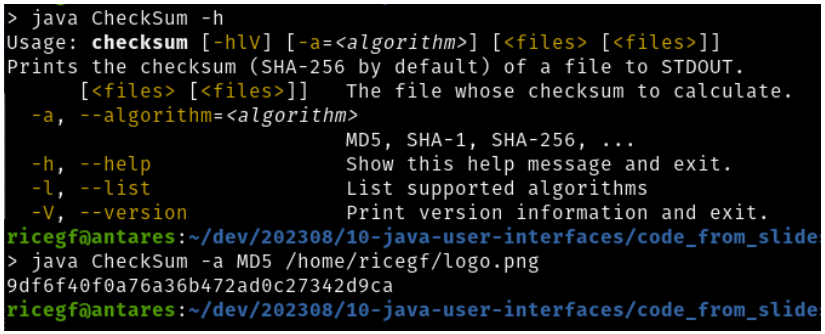
```
ricegf@antares:~/dev/cse1325-prof/P05/baseline$ git add .
ricegf@antares:~/dev/cse1325-prof/P05/baseline$ git commit -m 'P05 baseline (menu package)'
[main 82a669f] P05 baseline (menu package)
 8 files changed, 904 insertions(+)
 create mode 100644 P05/baseline/DemoMenu.java
 create mode 100644 P05/baseline/README.md
 create mode 100644 P05/baseline/build.xml
 create mode 100644 P05/baseline/menu/Menu.java
 create mode 100644 P05/baseline/menu/Menu.svg
 create mode 100644 P05/baseline/menu/Menu.uml
 create mode 100644 P05/baseline/menu/MenuItem.java
 create mode 100644 P05/baseline/menu/package-info.java
ricegf@antares:~/dev/cse1325-prof/P05/baseline$ git push
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 12 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (13/13), 9.75 KiB | 9.75 MiB/s, done.
Total 13 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/prof-rice/cse1325-prof.git
   97eab72..82a669f  main -> main
ricegf@antares:~/dev/cse1325-prof/P05/baseline$ dev
```

Command Line or Menu-Driven
C / C++, Java, Python, Rust

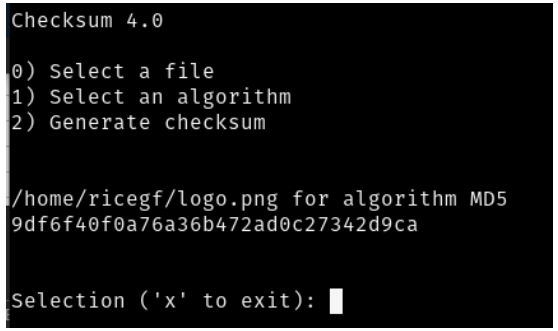
Virtual Reality
C++ and C#

Options for Writing Desktop User Interfaces in Java

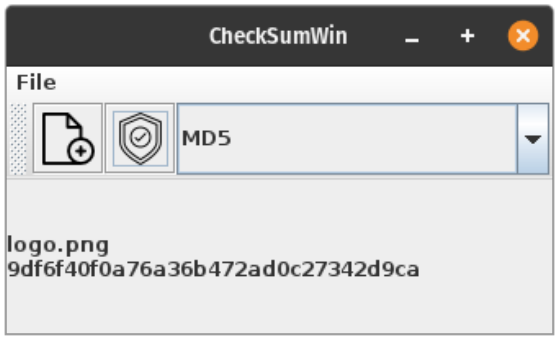
- Java fully supports 3 common desktop options
  - **Command Line Interface (CLI)**
    - This uses flags (-h or /h) and arguments (hello.java) to execute a single command with limited interaction
    - The **picoCLI** library is the best option



  - **Menu Driven Interface (MDI)**
    - This presents multiple levels of menus to build up and execute commands
    - Often hand-coded with a menu package

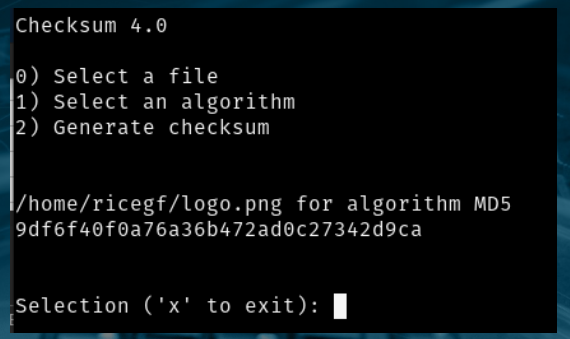


  - **Graphical User Interface (GUI)**
    - This present pull-down menus and buttons with dialogs to build up and execute commands
    - The **Swing** library is the best option



Hand-Coded vs Object-Oriented Menu Driven Interfaces

- You've written simple menu-driven programs
  - Use **`System.out.println`** to print each menu or sub-menu
  - **`System.util.Scanner`** your user's selection
  - Call a method to interact with the user for data and to implement your program's functionality

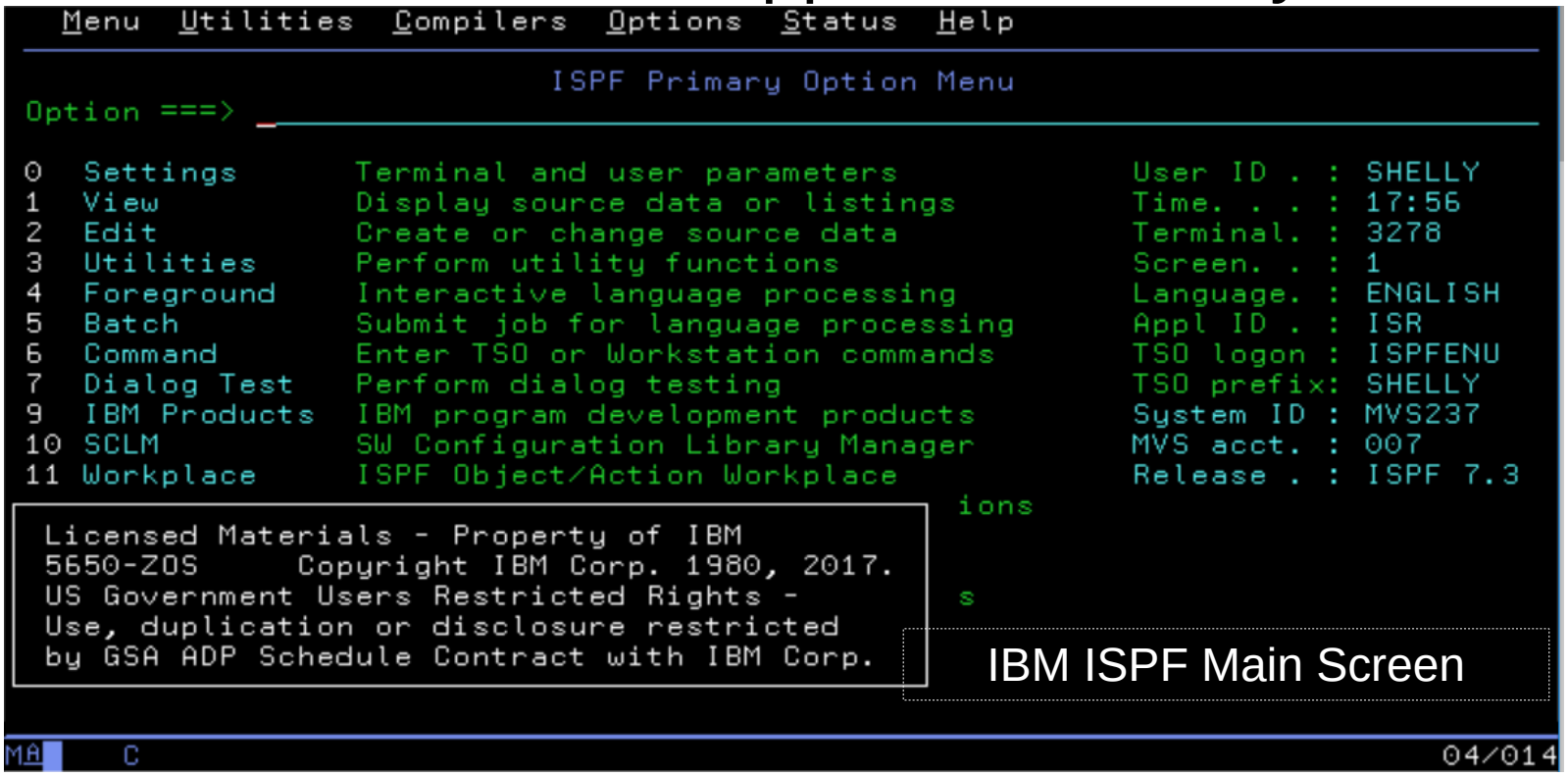


- Let's first examine a visual *pattern* for Object-Oriented Menu-Driven Interfaces
- Then we'll examine writing it OOP-style!

What Interface Do We Want?

Example: IBM ISPF Main Screen

- I recommend the common “forms” pattern
  - Menu at top, data in middle, prompt at the bottom
  - This was very common in the Age of Mainframes and still used in some applications today



What Interface Do We Want?

Example: Midnight Commander File Manager

Midnight Commander File Manager

What Interface Do We Want?

Example: Linux nano Text Editor

```
  GNU nano 5.4                                  EclecticMenu.java
import java.io.File;
import java.util.Scanner;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class EclecticMenu {
    private String title;
    private String output;
    private ArrayList<Object> stuff;
    private Scanner in = new Scanner(System.in);

    public EclecticMenu(String title) {
        this.title = title;
        this.stuff = new ArrayList<>();
        this.output = "";
    }

    // Method mdi() runs the menu system
    public void mdi() {
        while(true) {
            try {
                // Select an item from the menu
                Character c = selectFromMenu();
                if(c == null) continue;
                boolean valid = false;

                // Execute the handler for the selected item
                if(c == '1') {valid = true; listAllItems();  }
                if(c == '2') {valid = true; sortAllItems();  }
                if(c == '3') {valid = true; moveItem();      }
                                            [ Read 226 lines ]
^G Help         ^O Write Out    ^W Where Is     ^K Cut          ^T Execute      ^C Location     M-U Undo
^X Exit         ^R Read File    ^\ Replace      ^U Paste        ^J Justify      ^_ Go To Line   M-E Redo
```

nano Text Editor

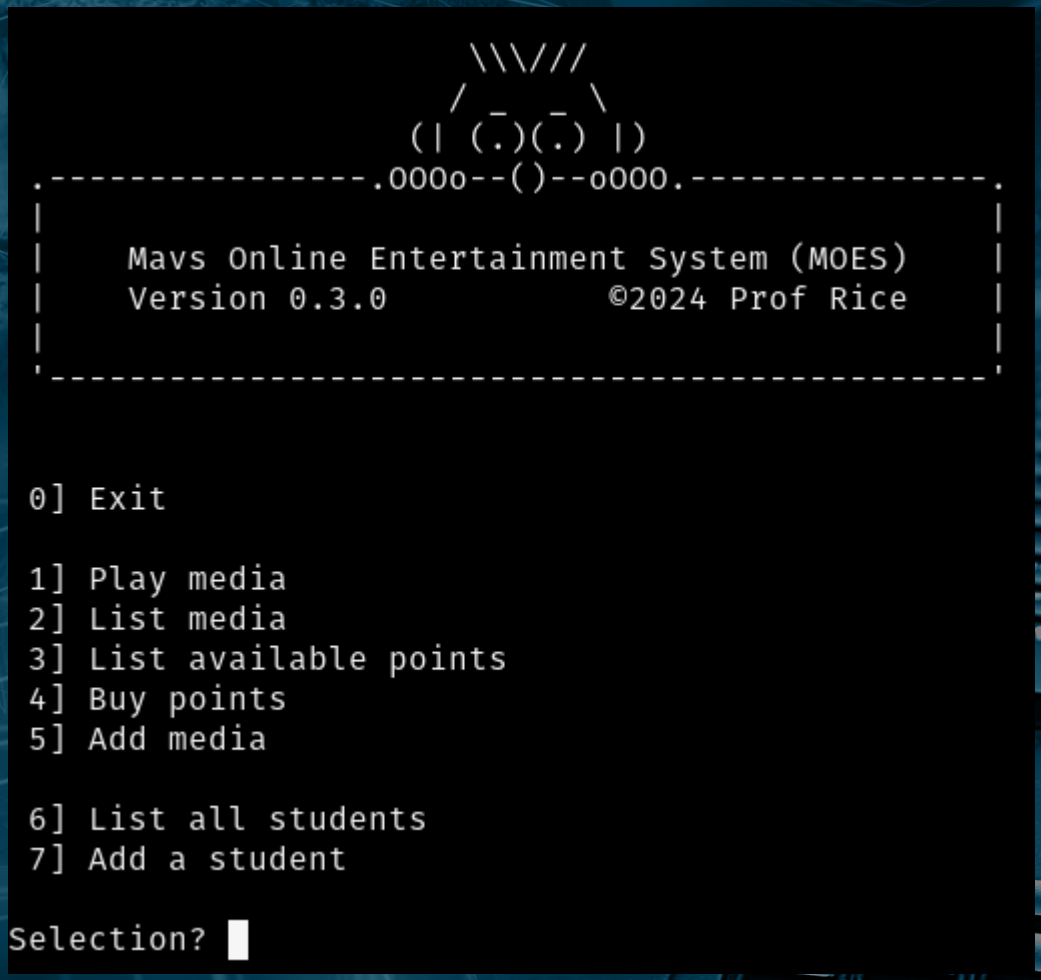
Yes, the menu is at the bottom here – c'est la vie – but the principle remains.

And yes, Microsoft is considering adding a terminal editor to Windows!

https://www.windowscentral.com/software-apps/microsoft-is-exploring-adding-a-command-line-text-editor-into-windows-and-it-wants-your-feedback

Menu-Driven Interface in Java

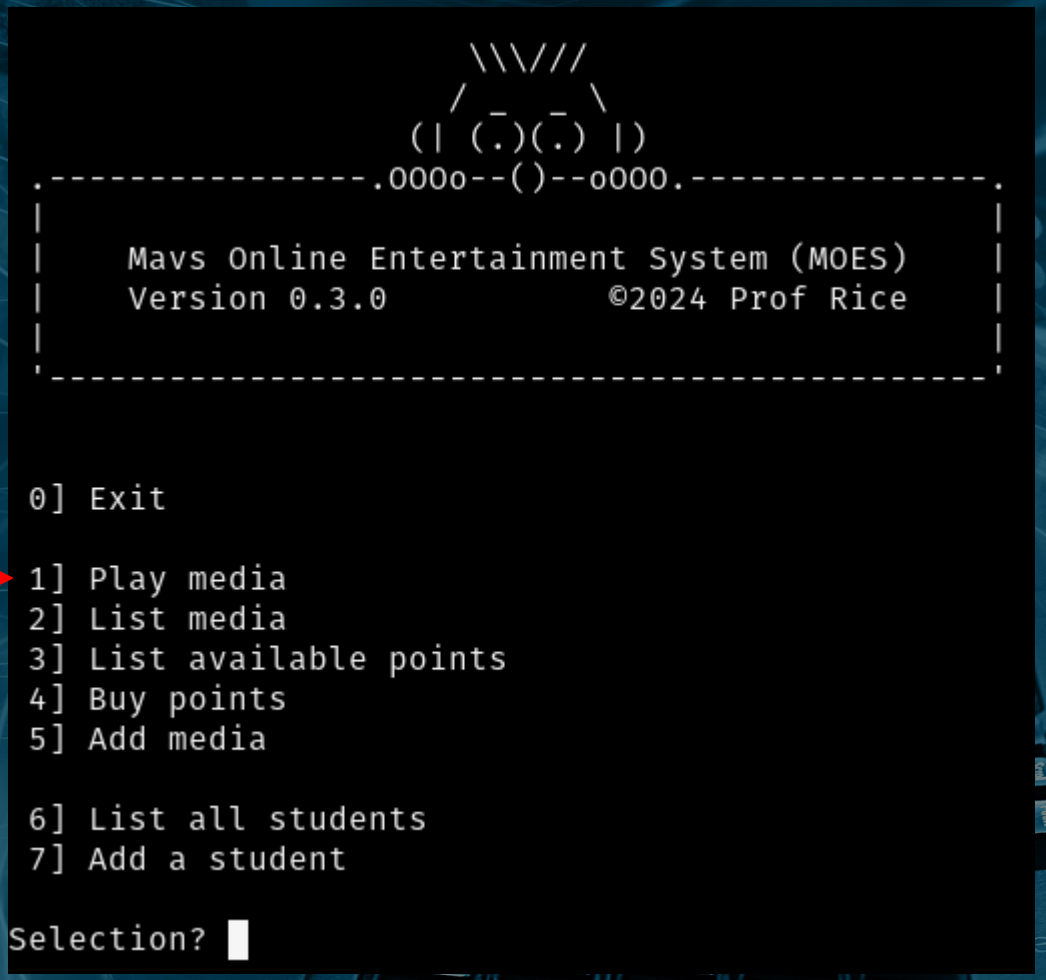
What Classes Do You See?



Menu-Driven Interface in Java

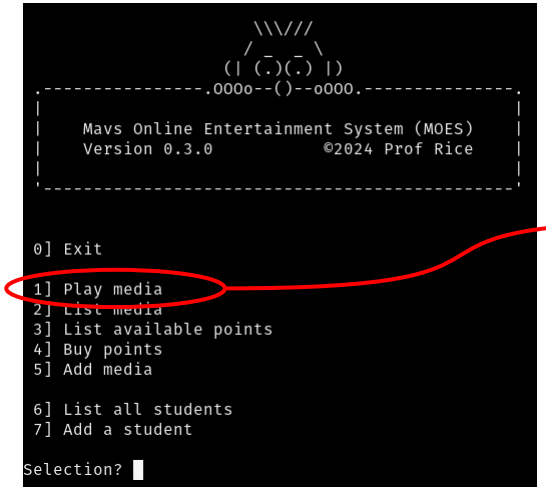
What Classes Do You See?

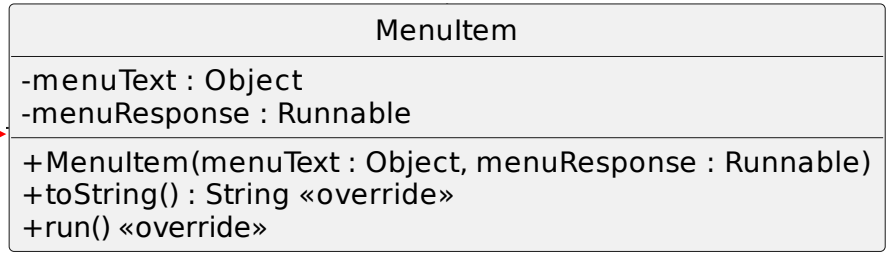
Each menu item could encapsulate both the text and the method to be called when that menu item is selected!



MenuItem

- Each item in the menu offers two potential fields
  - The menu text to show as one selection in the menu
  - The method to call when that menu item is selected
- **`MenuItem`** is thus an encapsulated menu text and method
  - The menu text is returned by toString() and printed to describe the behavior
  - The **`run()`** method implements the behavior when called

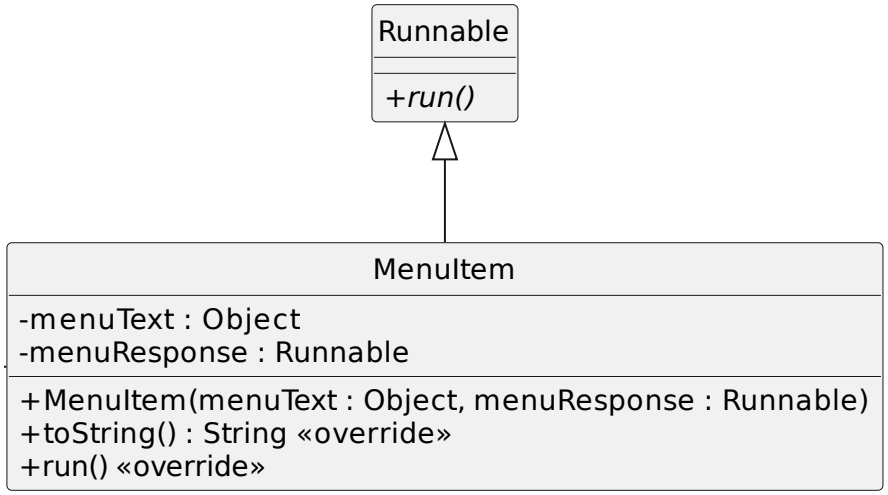




We'll defer the "1]" selector to a different class. Stay with me!

MenuItem

- In Java, methods are NOT first-class objects – we can't pass a method
  - But we CAN pass an *object* that has a *pre-defined* method!
- Use of this approach using a **`run()`** method is so common *it's in the library*!
  - It's described in a standard interface – **`Runnable`**
  - So we should implement that interface so we can *also* potentially **`ArrayList<? extends Runnable>`**



MenuItem

- The menu item's text is the **`toString()`** of the Object
  - Allowing any Object provides maximum flexibility
- The Runnable-required **`run()`** method chains to the menuResponse object's **`run()`** method

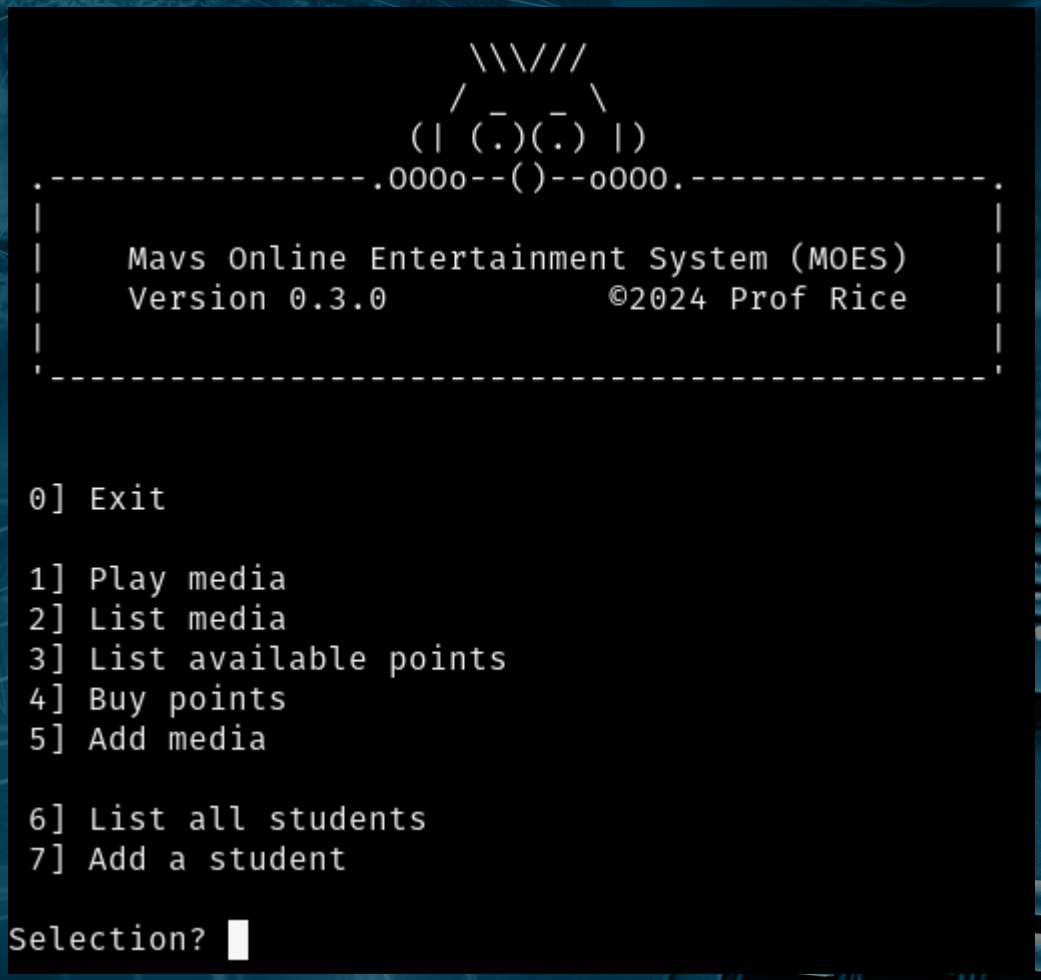
```
public class MenuItem implements Runnable {
    public MenuItem(Object menuText, Runnable menuResponse) {
        this.menuText = menuText;
        this.menuResponse = menuResponse;
    }
   @Override
    public String toString() {
        return menuText.toString();
    }
    @Override
    public void run() {
        menuResponse.run();
    }
    private Object menuText;        // The text displayed to the user
    private Runnable menuResponse;  // When selected, call this method via Runnable.run()
}
```

The Runnable interface requires a run() method

Example Instance:

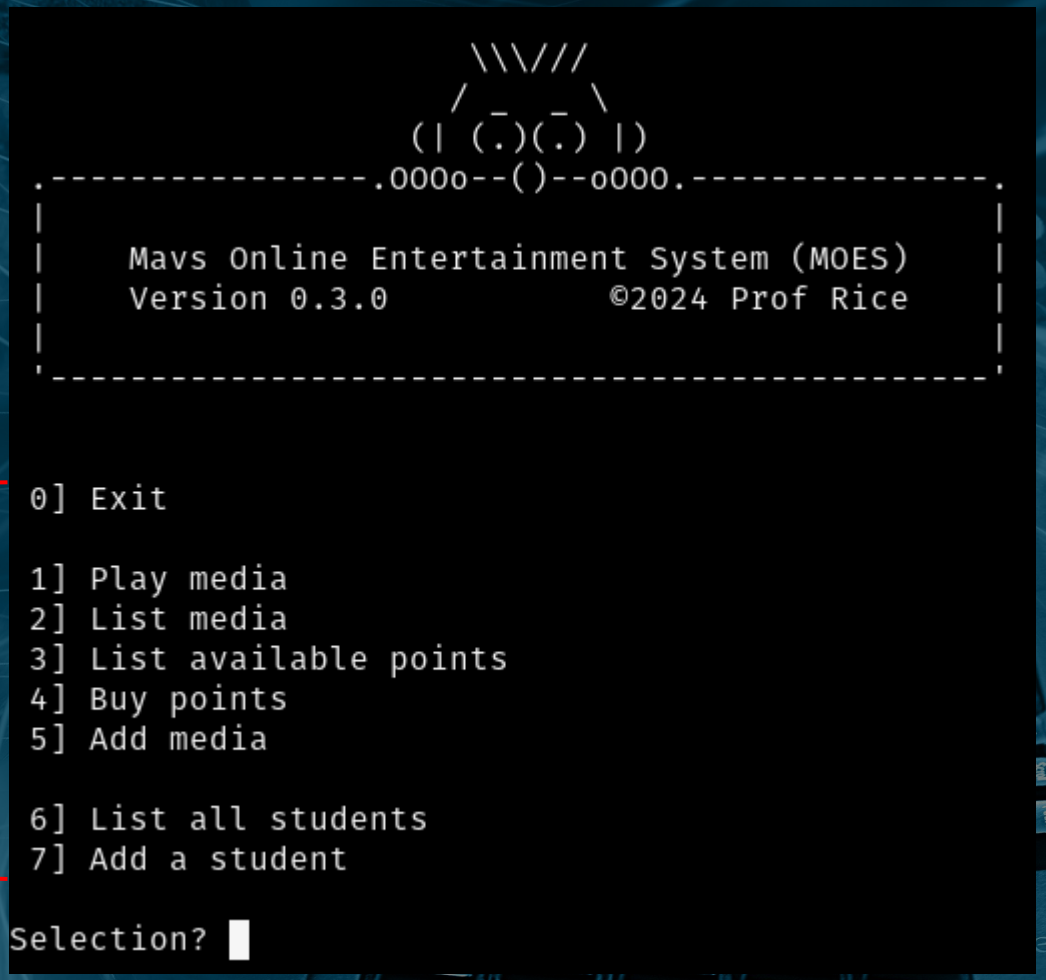
```
new MenuItem("Add an integer", () -> addInt()));
```

What Other Classes Do You See?



Menu!

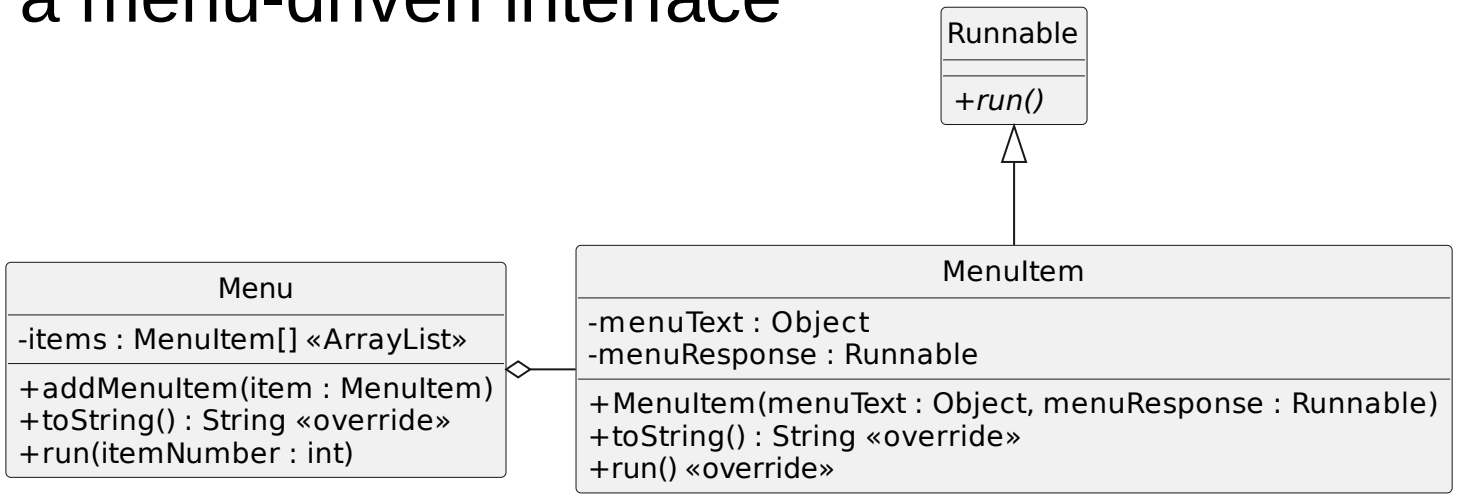
The menu itself is also an obvious class – it encapsulate an array (or better, a List) of MenuItem objects!



The Menu Class

- **`Menu`** is an encapsulated ArrayList of **`MenuItem`** (or **`? Extends Runnable`**) objects
  - Its toString() method prints each MenuItem's index and text
  - A **`run`** method that calls the associated method using its index
- This is a more object-oriented (and more maintainable) approach to a menu-driven interface

Menu.toString() prints the menu!



A Simple Menu Class

Collect the MenuItem Objects in an ArrayList

- Just collect your MenuItems in a Menu class with
  - A toString method (to format the entire menu with indices)
  - A dispatch mechanism (**`run(int index)`** method)!

```
public class Menu {
    private ArrayList<MenuItem> items = new ArrayList<>();

    public void addMenuItem(MenuItem item) {
        items.add(item);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for(int index=0 ; index<items.size() ; ++index)
            sb.append(" " + index + "] " + items.get(index) + "\n");
       return sb.toString();
    }

    public void run(int index) {
        items.get(index).run();
    }
}
```

Items contains pairs of menu text and their associated run() methods

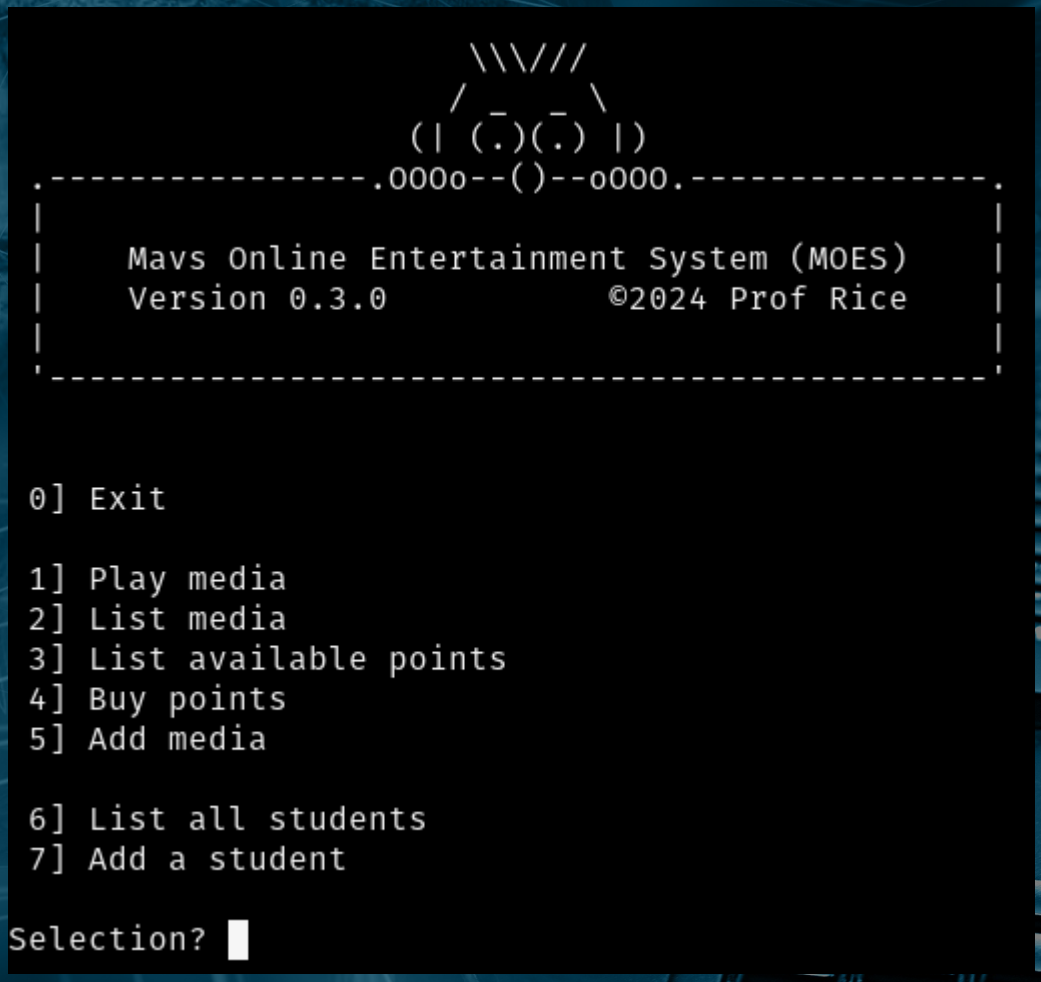
Add a menu item with associated run() method

Print the menu as index] menuItem.toString()

Call the associated method for index

Here's the MenuItem selector!

What Utility Methods Might We Need?



That is, how can we practice DRY
by encapsulating how to read common data types from the user?

Static Utility Methods
(Some Simple Thoughts)

- Break out “behaviors” and I/O “primitives” into methods including
  - **`String getString(prompt)`** – prints prompt, returns line of text typed by user, optional default text, handles all errors / retries, signals “Cancel” by returning null
  - **`Integer getInt(prompt)`** – gets String (ahem) and returns an Integer (or null)
    - Similar methods for Double, Character, and Boolean as well as application-specific types
  - **`selectItemFromList(prompt, List)`** – Displays items in List (such as ArrayList) next to their corresponding index, then gets the selected index with **`getInt`**
  - **`selectItemFromArray(prompt, Object[])`** – A similar method for arrays
  - **`void selectFromMenu()`** - prints menu and invokes the selected MenuItem
  - **`BufferedReader getReadFile(prompt)`** – prints prompt, opens a file for reading (next lecture!), with similar method for a **`BufferedWriter`**
- Behavior methods are usually application-specific
  - Although the common “About” (version and copyright) and “Help” menu items are potentially reusable

Not yet in our **`menu`** package

Reading a String

- What requirements should we support for **`String getString`**?

Reading a String
(Some *Grander* Thoughts)

- What requirements should we support for **`String getString`**?
  - Accept a prompt to print requesting a string
  - What input should indicate “cancel”, that is, we’d like to abort the entire current behavior such as “create a new Student”?
  - Should we accept a default input? This is useful when editing a composite object (just change the Student’s account type, for example, keeping the current name and student ID)
  - Should we provide a link to additional help?
  - What other features would you like to see?

Reading a String

- Here's our menu package's *grander* implementation
  - Yours may vary (if you like)!

Static methods
Menu.getString

```
// Show the prompt and return a string, with optional defaultInput
// If the user enters an empty string and defaultInput is not null, defaultInput is returned
// If cancelInput is not null and the user entered the cancelInput string, null is returned
public static String getString(String prompt, String cancelInput, String defaultInput) {
    String s = null;
    while(true) {
        try  {
            if(prompt != null) System.out.print(prompt);
            s = in.nextLine().trim();
            if(s.isEmpty() && defaultInput != null) s = defaultInput;
            break;
        } catch(Exception e) {
            System.err.println("Invalid input!");
        }
    }
    if(cancelInput != null && s.equals(cancelInput)) s = null;
    return s;
}
public static String getString(String prompt, String cancelInput) {
    return getString(prompt, cancelInput, null);
}
public static String getString(String prompt) {
    return getString(prompt, null, null);
}
```

Optional prompt.

Optional default value (just press Enter)

Optional cancel (returns null)

The Java approach to "optional" parameters (for space reasons, only the last method is shown on the upcoming class diagram).

Reading an Integer

- Here's a possible *grander* implementation
  - Yours may vary!

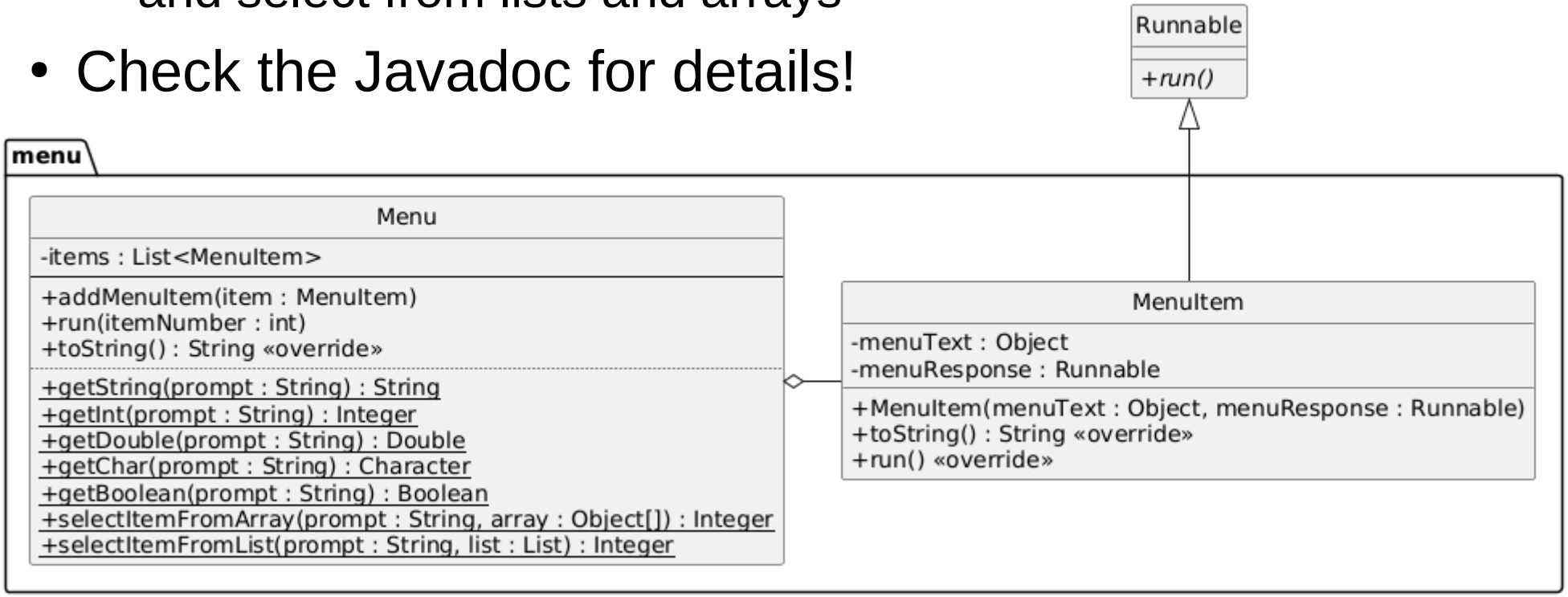
```
// Show the prompt and return an Integer (or null if either cancelInput is entered
//     or if an empty string is entered and defaultInput is null or omitted)
public static Integer getInt(String prompt, String cancelInput, String defaultInput) {
    Integer i = null;
    while(true) {
        try  {
            String s = getString(prompt, cancelInput, defaultInput);
            if(s != null && !s.isEmpty()) i = Integer.parseInt(s);
            break;
        } catch(Exception e) {
            System.err.println("Invalid input!");
        }
    }
    return i;
}
public static Integer getInt(String prompt, String cancelInput) {
    return getInt(prompt, cancelInput, null);
}
public static Integer getInt(String prompt) {
    return getInt(prompt, null, null);
}
```

Chain to getString
(a popular choice!)

Package menu

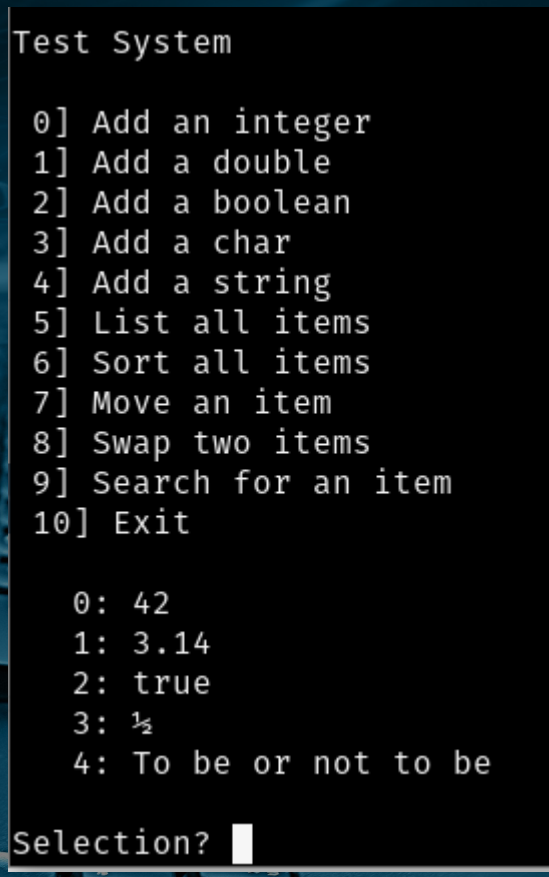
- So this is the full (thus far) menu package
  - MenuItem encapsulates each line of the menu
  - Menu (non-static) manages each menu
  - Static utilities quickly collect Strings and primitives and select from lists and arrays
- Check the Javadoc for details!

This makes P05 relatively straightforward!



OO MDI Example

- We'll create an app called EclecticMenu
  - Add any primitive or String to a List
  - Manipulate the List
    - List all items with their index
    - Sort the list by String representation
    - Move an item or swap two items' positions
    - Search for items containing specified text
- Not incredibly useful
  - But good enough as an example of writing an MDI application



Each Behavior is a Method

How to best invoke these from a menu?

```
private void addInt() {
    Integer i = Menu.getInt("Enter an integer to add to your stuff: ");
    if(i != null) {stuff.add(i); listAllItems();}
}

private void addString() {
    String s = Menu.getString("Enter text to add to your stuff: ");
    if(s != null) {stuff.add(s); listAllItems();}
}

private void listAllItems() {
    for(int i=0; i<stuff.size(); ++i)
        print(String.format("%4d: %s", i, stuff.get(i)));
}
private void sortAllItems() {
    Collections.sort(stuff, (Object a, Object b) -> a.toString().compareTo(b.toString()));
    listAllItems();
}

private void searchForItem() {
    String toFind = Menu.getString("Enter text to find: ");
    for(int i=0; i<stuff.size(); ++i)
        if(stuff.get(i).toString().contains(toFind))
            print(String.format("%4d: %s", i, stuff.get(i)));
}
```

EclecticMenuItems.java

Each “behavior” (or feature) is a separate, independent method that’s easier to code and to maintain.

This is a “lambda” – more details below and in Lecture 14!

The other behaviors are omitted here, but are available in the full application on cse1325-prof.

Here's the "C" Way to do MDI

Independent menu and dispatch isn't very maintainable

```
// Show the main menu and return the char selected
private static final String clearScreen = "\n".repeat(255);
private Character selectFromMenu() {
    System.out.println(clearScreen + title + '\n');
    System.out.println("i) Add an integer)";
    System.out.println("d) Add a double     1) List all items");
    System.out.println("b) Add a boolean    2) Sort all items");
    System.out.println("c) Add a char       3) Move an item");
    System.out.println("s) Add a string     4) Swap two items");
    System.out.println("x) Exit             5) Search for an item");
    System.out.println();
    System.out.println(output);
    output = "";
    return getChar("Selection? ");
}
public void mdi() {
    while(true) {
        try {
            // Select an item from the menu
            Character c = selectFromMenu();
            if(c == null) continue;
            boolean valid = false;

            // Execute the handler for the selected item
            if(c == '1') {valid = true; listAllItems();  }
            if(c == '2') {valid = true; sortAllItems();  }
            if(c == '3') {valid = true; moveItem();      }
            if(c == '4') {valid = true; swapTwoItems();  }
            if(c == '5') {valid = true; searchForItem(); }

            if(c == 'i') {valid = true; addInt();        }
            if(c == 'd') {valid = true; addDouble();     }
            if(c == 'b') {valid = true; addBoolean();    }
```

Maintenance Headache

This is a LOT of non-synced, custom code.

Let's try the OO way instead!

Syncing Menu and Dispatch
A Better Way

- The constructor now builds the Menu as a simple, *executable* table – *much* easier to maintain!

The constructor!

The *lambda* converts a method into a Runnable object (see later slide)

```
public EclecticMenuItems(String title) {
    this.menu = new Menu();

    menu.addMenuItem(new MenuItem("Exit",                () -> endApp()));
    menu.addMenuItem(new MenuItem("Add an integer",      () -> addInt()));
    menu.addMenuItem(new MenuItem("Add a double",        () -> addDouble()));
    menu.addMenuItem(new MenuItem("Add a boolean",       () -> addBoolean()));
    menu.addMenuItem(new MenuItem("Add a char",          () -> addChar()));
    menu.addMenuItem(new MenuItem("Add a string",        () -> addString()));
    menu.addMenuItem(new MenuItem("List all items",      () -> listAllItems()));
    menu.addMenuItem(new MenuItem("Sort all items",      () -> sortAllItems()));
    menu.addMenuItem(new MenuItem("Move an item",        () -> moveItem()));
    menu.addMenuItem(new MenuItem("Swap two items",      () -> swapTwoItems()));
    menu.addMenuItem(new MenuItem("Search for an item", () -> searchForItem()));
}
```

Notice that selecting “Exit” will call endApp()

If they select this call this !
Now the dispatch table *looks* like a dispatch table!

Syncing Menu and Dispatch
A Better Way

- The main loop simply gets an int (the MenuItem index) and dispatches the associated method

```
// ///////////////////////////////////////////////////////////////
//                          M A I N   M E N U

public void mdi() {
    while(running) {
        try {
            Integer i = selectFromMenu();
            if(i == null) continue;
            menu.run(i);
        } catch (Exception e) {
            print("#### Invalid command");
        }
    }
}

private static boolean running = true;
private void endApp() {
    running = false;
}
```

The main loop is short, sweet, and *reusable*!
It will work in all of your MDI programs unmodified.

```
Test System

 0] Add an integer
 1] Add a double
 2] Add a boolean
 3] Add a char
 4] Add a string
 5] List all items
 6] Sort all items
 7] Move an item
 8] Swap two items
 9] Search for an item
 10] Exit

   0: 42
   1: 3.14
   2: true
   3: ½
   4: To be or not to be

Selection?
```

When the user selected “Exit”, menu.run(i) invokes endApp(). endApp() sets running to false, causing the main loop to exit.

Displaying and Selecting from the Menu

- Menu interaction is just a few lines of code
- This may be a separate method or in-line within your main loop

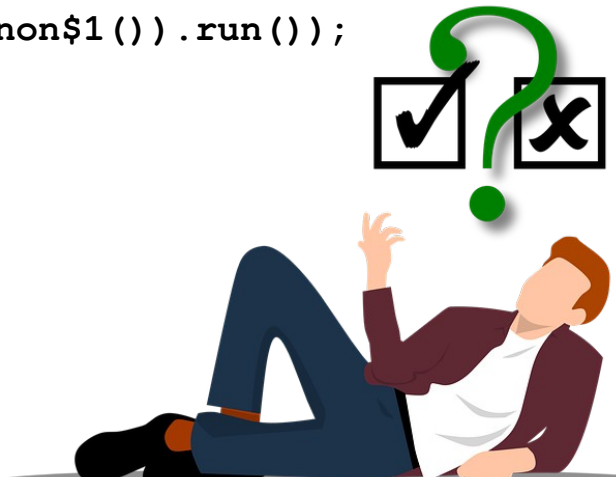
```
// ////////////////////////////////////////////////////////////////////////
//                          M A I N   M E N U

// Show the main menu and return the char selected
private static String clearScreen = "\n".repeat(255);
private Integer selectFromMenu() {
    System.out.println(clearScreen + title + '\n' + menu + '\n' + output);
    output = "";
    return getInt("Selection? ");
}
```

```
Test System

 0] Add an integer
 1] Add a double
 2] Add a boolean
 3] Add a char
 4] Add a string
 5] List all items
 6] Sort all items
 7] Move an item
 8] Swap two items
 9] Search for an item
 10] Exit

   0: 42
   1: 3.14
   2: true
   3: ½
   4: To be or not to be

Selection?
```

Lambda

- A **lambda** is a an anonymous *method* object.
  - Basically, you specify an expression and get an entire class implementing an interface and instanced into an object for *free*!
- It is usually defined where it is invoked
- It is most suitable for implementing interfaces that require a single method, especially if the result is an expression


```
new MenuItem("x",(new Anon$1()).run());

public class Anon$1
  implements Runnable {
    @Override
    public void run() {
        addInt();
    }
}
```



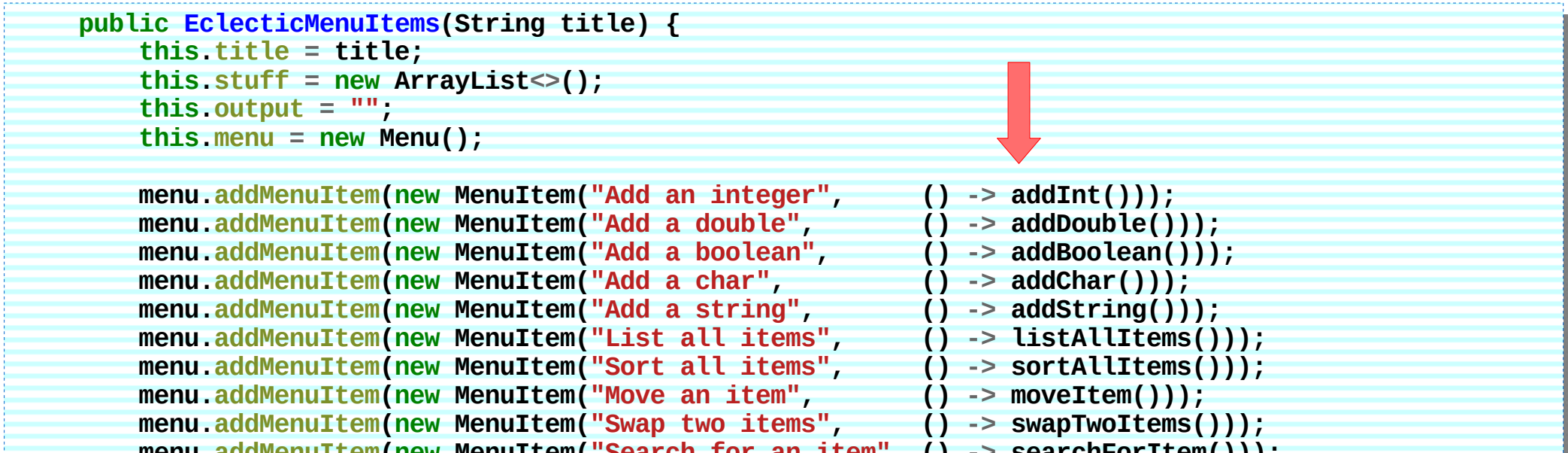
```
new MenuItem("x",() -> addInt());
```

Tough choice, huh?

Decision image by booger_picker per the Pixabay License
https://pixabay.com/vectors/decide-decision-choice-choose-vote-6867641/

Lambda

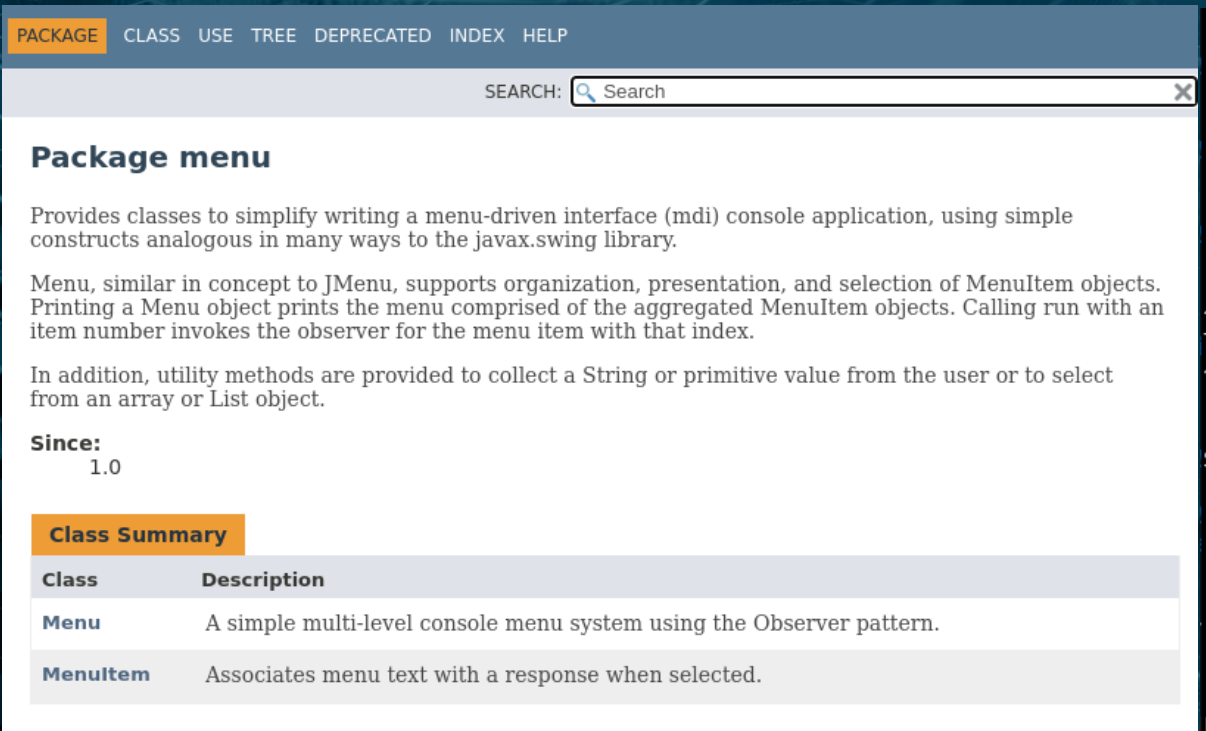
- A lambda consists of
  - A comma-separated parameter list without the types
    (and, for a single parameter method, without the parentheses, too!)
  - The arrow ->
  - The body
    - If a single expression, the result is the return value
    - If multiple statements, enclose in { } and use a return statement

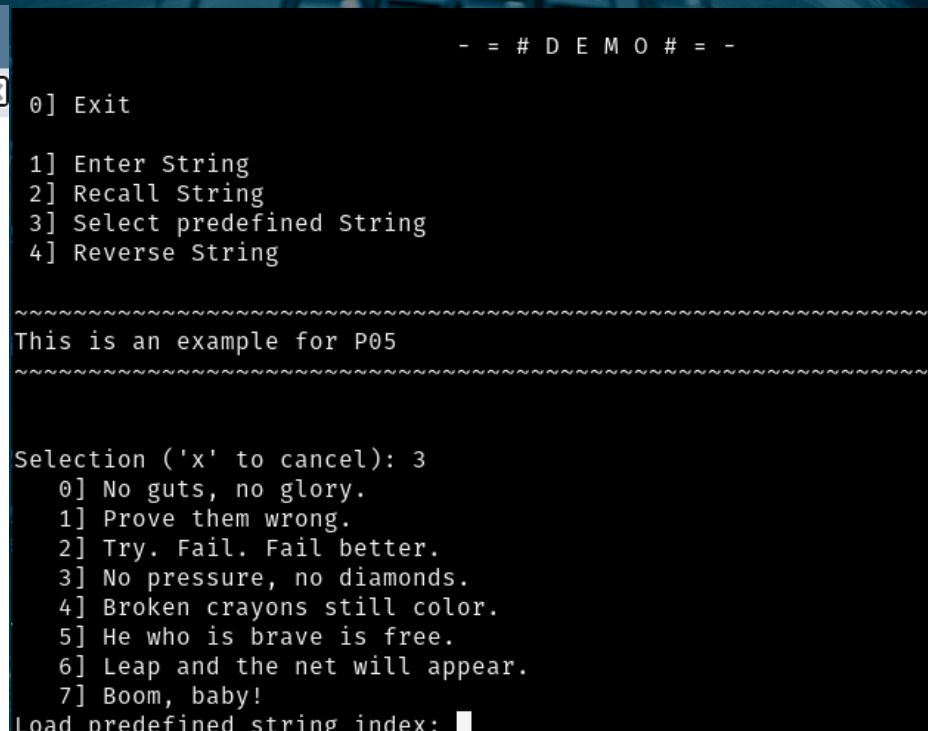


```
public EclecticMenuItems(String title) {
    this.title = title;
    this.stuff = new ArrayList<>();
    this.output = "";
    this.menu = new Menu();

    menu.addMenuItem(new MenuItem("Add an integer",     () -> addInt()));
    menu.addMenuItem(new MenuItem("Add a double",       () -> addDouble()));
    menu.addMenuItem(new MenuItem("Add a boolean",      () -> addBoolean()));
    menu.addMenuItem(new MenuItem("Add a char",         () -> addChar()));
    menu.addMenuItem(new MenuItem("Add a string",       () -> addString()));
    menu.addMenuItem(new MenuItem("List all items",     () -> listAllItems()));
    menu.addMenuItem(new MenuItem("Sort all items",     () -> sortAllItems()));
    menu.addMenuItem(new MenuItem("Move an item",       () -> moveItem()));
    menu.addMenuItem(new MenuItem("Swap two items",     () -> swapTwoItems()));
    menu.addMenuItem(new MenuItem("Search for an item", () -> searchForItem()));
```

Get the menu Package!

- It's available in cse1325-prof at P05/baseline
- FULL Javadoc documentation is provided
  - In that directory: **`ant javadoc`** and open docs/api/index.html
- DemoMenu.java is another menu example
  - In that directory: **`ant ; java DemoMenu`**






The End

Bonus Section

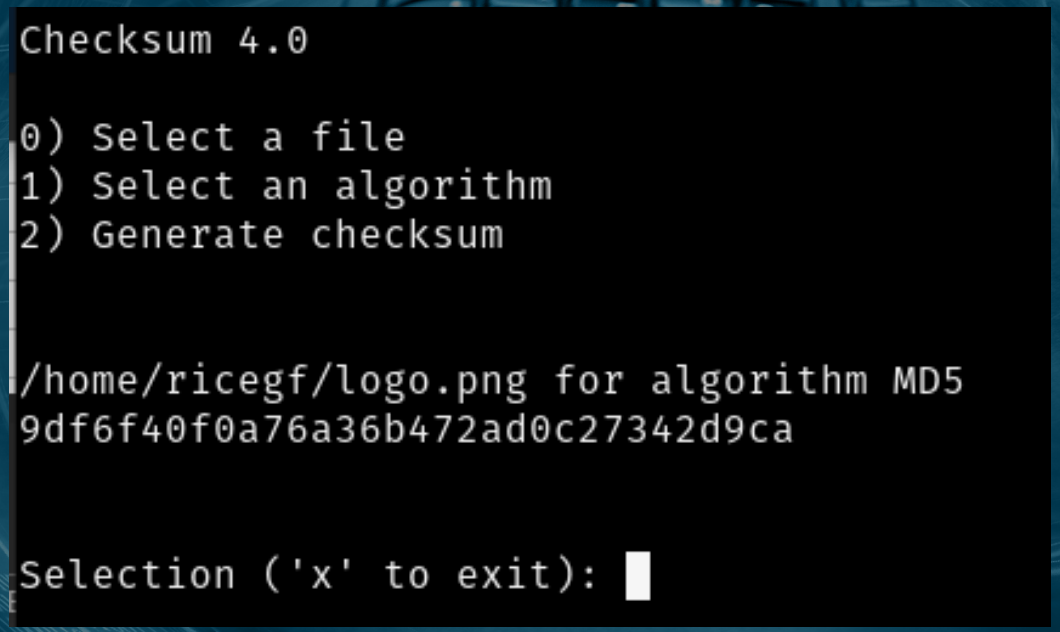
Comparing Other Interface Options

- Let’s write another utility using 5 different approaches
  - Menu-Driven Interface – hand-crafted and attomenu
  - Command Line Interface – hand-crafted and picoCLI
  - Graphical User Interface using Java Swing
- Compare ease of coding and usability
- Example program: **Checksum 4.0**
  - Generate file checksums using a pre-installed algorithm on your own system
  - List and select from all pre-installed algorithms
  - Conveniently select a file and show its checksum

Option 1

Hand-Coded Menu Driven Interface

- We'll use a nested class to hold our data
- We'll write a separate method to implement each menu item
- We'll hard-code the menu and dispatch table logic



Menu: Data and a Listener

Utility to Generate File Checksums

```
import java.io.File;
import java.math.BigInteger;
import java.nio.file.Files;
import java.security.MessageDigest;
import java.security.Security;

public class CheckSumMenu {
    private static String noChecksum = "\n";
    class Data {
        public File file = null;              // The file whose checksum to calculate
        public String algorithm = "SHA-256"; // Checksum algorithm name
        public String checksum = noChecksum; // Calculated checksum
        @Override
        public String toString() {           // How to present this data to the user
            return (file == null ? "No file selected" : file.toString())
                    + " for algorithm " + algorithm + checksum;
        }
    }
    private Data data = new Data();

    // Listeners
    protected void onAddFile() {
        String filename = "";
        try {
            filename = System.console().readLine("Select file: ");
            data.file = new File(filename);
            data.checksum = noChecksum;
        } catch (Exception e) {
            data.checksum = "\nInvalid filename: " + filename + "\n" + e + "\n";
        }
    }
```

Here's our simple utility in a hand-crafted menu form. Note that we organize our *data* into a nested class to make formatting easier, and break out *behaviors* (a "thing" the program does) into separate methods.

Data.toString() formats our data as we want to present it to our end users.

"Listeners" react to menu selections

Selecting a file is simplified here but user-hostile. Who want's to type in the fully qualified path?

Continued on next page...

Menu: Remaining Listeners

Utility to Generate File Checksums

```
protected void onSelectAlgorithm() {
    try {
        String[] algorithms = Security.getAlgorithms("MessageDigest").toArray(new String[0]);
        for(int i=0; i<algorithms.length; ++i)
            System.out.println("" + i + ") " + algorithms[i]);
        String s = System.console().readLine("Select checksum algorithm ('x' to cancel): ");
        if(s.charAt(0) == 'x') return;
        data.algorithm = algorithms[Integer.parseInt(s)];
        data.checksum = noChecksum;
    } catch(Exception e) {
        System.err.println("Unable to select an algorithm: " + e);
    }
}

protected void onGenerateChecksum() {
    try {
        if(data.file != null) {
            byte[] fileContents = Files.readAllBytes(data.file.toPath());
            byte[] digest = MessageDigest.getInstance(data.algorithm).digest(fileContents);
            data.checksum = String.format("\n%0" + (digest.length*2) + "x",
                new BigInteger(1, digest));
        }
    } catch(Exception e) {
        data.checksum = "Failed to generate checksum: " + e;
    }
}
```

Security.getAlgorithms returns a String array listing all that are supported at runtime.

Since we don't know how long checksum calculations will take, we defer until specificallyrequested.

Do I even *need* to say this isn't on the exam? *Certainly* I looked it up and adapted code!

Continued on next page...

Menu: Main Loop

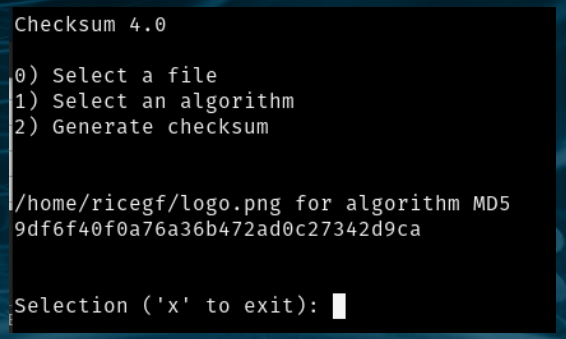
Utility to Generate File Checksums

```
    public static void main(String... args) {
        CheckSumMenu menu = new CheckSumMenu();
        while(true) {
            try {
                System.out.println("\n\n\n\n\n\n\n\nChecksum Menu 4.0\n");
                System.out.println("0) Select a file");
                System.out.println("1) Select an algorithm");
                System.out.println("2) Generate checksum");
                System.out.println("\n" + menu.data + "\n");
                String s = System.console().readLine("\nSelection ('x' to exit): ");
                if(s.charAt(0) == 'x') break;
                int selection = Integer.parseInt(s);
                if(selection == 0) menu.onAddFile();
                else if(selection == 1) menu.onSelectAlgorithm();
                else if(selection == 2) menu.onGenerateChecksum();
                else throw new IllegalArgumentException(s + " - select between 0 and 2 or x");
            } catch (Exception e) {
                System.err.println("Invalid selection: " + e);
            }
        }
    }
}
```

Note that we define the menu and associated behaviors separately

Here we laboriously print out and manage a menu, calling the behavior methods (called “listeners” or “observers” in professional circles).

Thoughts on Hand-Coding Menus



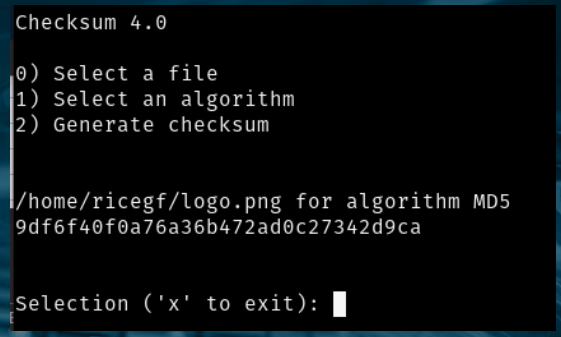
- This is 75 lines of Java code
  - Most of the code cannot be reused without editing
    - Editing code is NOT reusing code!
      - You keep the original *bugs* but can't sync bug *fixes* with the original code
      - You add *new* bugs through misunderstanding the code design
- The menus are independent
  - Main menu, selecting an algorithm are redundant
  - Typing in a full path and filename? Welcome to the '70s!
- We really want to specify all aspects exactly once
  - Don't Repeat Yourself (DRY)

Option 2

Attomenu Interfaces

(that is, small menu-driven interfaces)

- Some brilliant young prodigy (ahem) wrote a package to make this easy for you (well, easi-*er*)
  - Multi-level menus are quickly defined as (text, method) pairs
  - Formatting, error handling, and ‘x’ to exit are all effectively “free” (no coding on your part)
  - Static methods handle selection from an array or List and file system navigation and selection – including filtering (for specified extensions) and creating folders
  - Your code is much more organized and maintainable
  - Swing conventions are followed
    - So if you’re learning Swing, too, this will seem *very* familiar!



Attomenu: Data & Main Menu

Utility to Generate File Checksums

```
import java.io.File;
import java.math.BigInteger;
import java.nio.file.Files;
import java.security.MessageDigest;
import java.security.Security;

import attomenu.Menu;
import attomenu.MenuItem;

public class CheckSumAttoMenu {
    private static String noChecksum = "\n";
    class Data {
        public File file = null;                // The file whose checksum to calculate
        public String algorithm = "SHA-256"; // Checksum algorithm name
        public String checksum = noChecksum; // Calculated checksum
        @Override
        public String toString() {
            return (file == null ? "No file selected" : file.toString())
                    + " for algorithm " + algorithm + checksum;
        }
    }
    private Data data;
    private Menu menu;
    public CheckSumAttoMenu() {
        data = new Data();
        menu = new Menu("\n\n\n\n\n\n\n\nChecksum 4.0", data,
            new MenuItem("Select a file",       () -> onAddFile()),
            new MenuItem("Select an algorithm", () -> onSelectAlgorithm()),
            new MenuItem("Generate checksum",   () -> onGenerateChecksum()));
        menu.run();
    }
```

Import the classes we need from the attomenu package

As before, nested class Data stores user-selected info and formats it via toString

The constructor concisely defines our menu, which could include sub-menus (which we didn't need here). When the menu item is selected, call the method after () -> (this is a *lambda*!). Note that the menu item and associated behavior is *together*!

This is ALL of the menu code!

Menu.run() is the main loop

Continued on next page...

Attomenu: Listeners and main

Utility to Generate File Checksums

```
    // Listeners
    protected void onAddFile() {
        data.file = Menu.selectFile("Select File",                    // set the dialog's title
            (data.file != null) ? data.file.getParentFile() : null, // start in selected folder
            null);                                                    // show all files (no filter)
        data.checksum = noChecksum;
    }
    protected void onSelectAlgorithm() {
        String[] algorithms = Security.getAlgorithms("MessageDigest").toArray(new String[0]);
        int selection = Menu.select("Select checksum algorithm", algorithms);
        if(selection != -1) {
            data.algorithm = algorithms[selection];
            data.checksum = noChecksum;
        }
    }
    protected void onGenerateChecksum() {
        try {
            if(data.file != null) {
                byte[] fileContents = Files.readAllBytes(data.file.toPath());
                byte[] digest = MessageDigest.getInstance(data.algorithm).digest(fileContents);
                data.checksum = String.format("\n%0" + (digest.length*2) + "x",
                    new BigInteger(1, digest));
            }
        } catch(Exception e) {
            data.checksum = "Failed to generate checksum: " + e;
        }
    }
    public static void main(String... args) {
        new CheckSumAttoMenu();
    }
}
```

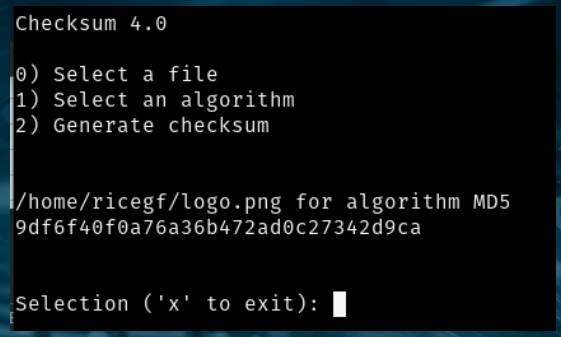
Single-line interactive selection of a file!

Single-line selection from the array of algorithms!

Just instance your main class – the main loop is free!

Thoughts on CheckSum with Attomenu

- This is 59 (not 75) lines of Java code – almost 25% less!
  - Less code means fewer bugs
    (and UI code tends to be buggy)
- We get a lot of “free” code
  - We can finally *navigate* the file system
    and *select* the file we want rather than typing in the full path!
  - The program remembers where we were on the next file selection
  - The main loop is free
  - Error checking, error message generation, and “try again” are free
- We specify all aspects exactly once
- Now you Don’t Repeat Yourself (DRY) as much



Using Attomenu

- Obtain a copy
  - From (easy) cse1325-prof/attomenu or (latest source)
    https://github.com/prof-rice/attomenu
- Check the documentation
  - https://prof-rice.github.io/attomenu/api/attomenu/package-summary.html
  - See attomenu/pizza is a more complete example
- In your constructor, declare each menu and submenu as
  `new Menu(Object title, Object data, MenuItem… menuItems)`
  - Each MenuItem is
    `new MenuItem(Object menuText, Runnable menuResponse)`
    - The menu item's text is **`menuText.toString()`**
    - menuResponse may be a submenu, perhaps **`subMenu.runOnce()`**
      or menuResponse may call a method in your class or another class
      or even be literal code: **`new MenuItem("Say hi", ()->System.out.println("Hi!));`**
  - In the menu, **`title.toString()`** precedes and **`data.toString()`** follows the menu item list
  - Any number of MenuItems will be accepted (that's what … means in Java parameter lists!)
- In your constructor, after defining your menus, just call **`menu.run()`** (**`menu`** is your main menu)
- Finally, in your main method, just instance your class – done and done!

Option 3

Hand-Coded Command Line Interfaces

- One run of the program per file
  - No need to store data – display as generated
  - One operation for each execution
- Must parse the command line arguments
- Will need a usage statement (similar to menu)
- Will need version info (similar to about)

```
> java CheckSum -h
Usage: checksum [-hlV] [-a=<algorithm>] [<files> [<files>]]
Prints the checksum (SHA-256 by default) of a file to STDOUT.
      [<files> [<files>]]   The file whose checksum to calculate.
  -a, --algorithm=<algorithm>
                            MD5, SHA-1, SHA-256, ...
  -h, --help                Show this help message and exit.
  -l, --list                List supported algorithms
  -V, --version             Print version information and exit.
ricegf@antares:~/dev/202308/10-java-user-interfaces/code_from_slide
> java CheckSum -a MD5 /home/ricegf/logo.png
9df6f40f0a76a36b472ad0c27342d9ca
ricegf@antares:~/dev/202308/10-java-user-interfaces/code_from_slide
```

Classic CLI: Usage & Error Reports

Utility to Generate File Checksums

```
import java.io.File;
import java.math.BigInteger;
import java.nio.file.Files;
import java.security.MessageDigest;
import java.security.Security;

class CheckSum {
    private String version = "4.0";
    private File file = null;
    private String algorithm = "SHA-256";

    private void usage() {
        System.err.println("Usage: checksum [-lhV] [-a=<algorithm>] <file>\n"
                         + "Prints the checksum (SHA-256 by default) of a file to STDOUT.\n"
                         + "      <file>      The file whose checksum to calculate.\n"
                         + "  -a, --algorithm=<algorithm>\n"
                         + "                  MD5, SHA-1, SHA-256, ...\n"
                         + "  -l, --list      Show all supported algorithms.\n"
                         + "  -h, --help      Show this help message and exit.\n"
                         + "  -V, --version   Print version information and exit.\n");
    }

    private void badArgument(int index, String argument) {
        if(argument.charAt(0) == '-')
            System.err.println("Unknown option: '" + argument + "'");
        else
            System.err.println("Unmatched argument at index " + index + ": '" + argument + "'");
        usage();
        System.exit(-1);
    }
```

The usage statement is hand-coded for brevity.

Continued on next page...

Classic CLI: Constructor

Utility to Generate File Checksums

```
public CheckSum(String[] args) {
    int i=0;
    while(i < args.length) {
        if(args[i].equals("-a") || args[i].equals("--algorithm")) {
            algorithm = args[++i];
            ++i;
        } else if(args[i].equals("-l") || args[i].equals("--list")) {
            listAlgorithms();
            System.exit(0);
        } else if(args[i].equals("-h") || args[i].equals("--help")) {
            usage();
            System.exit(0);
        } else if(args[i].equals("-V") || args[i].equals("--version")) {
            System.out.println(version);
            System.exit(0);
        } else if(file == null) {
            if(args[i].charAt(0) == '-') badArgument(i, args[i]);
            else file = new File(args[i++]);
        } else {
            badArgument(i, args[i]);
        }
    }

    if(file == null) {
        System.err.println("Missing required parameter: '<file>'");
        usage();
        System.exit(-2);
    }
}
```

Handling command line arguments takes quite a bit of code in the constructor, doesn't it?

Continued on next page...

Classic CLI: Listeners and main

Utility to Generate File Checksums

```
    public void listAlgorithms() {
        for(String s : Security.getAlgorithms("MessageDigest").toArray(new String[0]))
            System.out.println(s);
    }

    public void printCheckSum() throws Exception {
        byte[] fileContents = Files.readAllBytes(file.toPath());
        byte[] digest = MessageDigest.getInstance(algorithm).digest(fileContents);
        System.out.printf("%0" + (digest.length*2) + "x%n", new BigInteger(1, digest));
        System.exit(0);
    }

    public static void main(String[] args) throws Exception {
        (new CheckSum(args)).printCheckSum();
    }
}
```

Running CheckSum

Nothing special to build

Hand-coded error messages…

… and usage information…

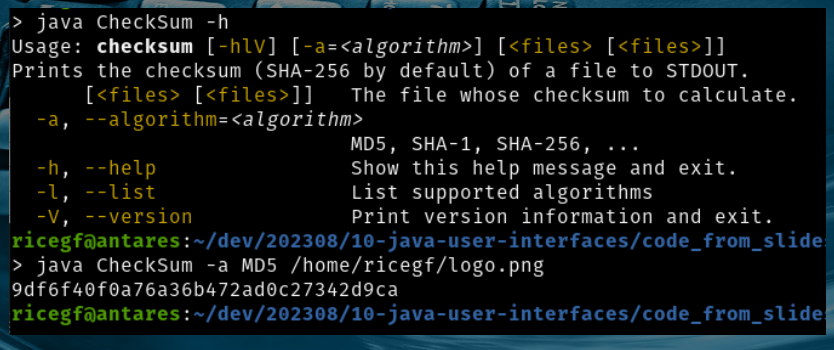
… and version information

Runs great!

```
ricegf@antares:~/dev/202308$ ls CheckSumClassic.java
CheckSumClassic.java
ricegf@antares:~/dev/202308$ javac CheckSumClassic.java
ricegf@antares:~/dev/202308$ java CheckSumClassic
Missing required parameter: '<file>'
Usage: checksum [-hV] [-a=<algorithm>] <file>
Prints the checksum (SHA-256 by default) of a file to STDOUT.
      <file>      The file whose checksum to calculate.
  -a, --algorithm=<algorithm>
                  MD5, SHA-1, SHA-256, ...
  -h, --help      Show this help message and exit.
  -V, --version   Print version information and exit.

ricegf@antares:~/dev/202308@ java CheckSumClassic --version
4.0
ricegf@antares:~/dev/202308$ java CheckSumClassic -a SHA-1 CheckSumC
d8158de10752e7fba6773e11e3479f79b2d2bee4
ricegf@antares:~/dev/202308$ java CheckSumClassic CheckSumClassic.ja
9f1d11b26293e0b0f54cfadc190d112536df0c7e8ce6af2aab8f693a1c1b60c0
ricegf@antares:~/dev/202308$
```

Thoughts on CLI CheckSum

- This is 69 lines of Java code (59 / 75 for menu)
  - Only the badArgument method is truly reusable
  - Constructor and usage method could be edited
    - But edited reuse saves a LOT less than unmodified reuse
- The usage statement is too independent
  - If you modify the constructor, you must remember to modify usage() to match
- We really want to specify all aspects exactly once
  - Don’t Repeat Yourself (DRY)



Option 4
picocli
(that is, small Command Line Interfaces)

- picocli is a single-file framework for creating Java CLI applications with minimal code
  - Supports flag / option styles for Linux, Windows, and others
  - Auto-generates colorful help and error messages
  - Auto-generates HTML / PDF docs and Linux man pages
  - Supports tab completion for bash, DOS, and PowerShell
  - Supports creating a single-file executable for distribution
- Simply use annotations similar to @override with parameters to describe your desired user interface

Picocli does a lot more than Attomenu – that's why it's *pico* and not just *atto*! :)

```
> java CheckSum -h
Usage: checksum [-hlV] [-a=<algorithm>] [<files> [<files>]]
Prints the checksum (SHA-256 by default) of a file to STDOUT.
      [<files> [<files>]]   The file whose checksum to calculate.
  -a, --algorithm=<algorithm>
                            MD5, SHA-1, SHA-256, ...
  -h, --help                Show this help message and exit.
  -l, --list                List supported algorithms
  -V, --version             Print version information and exit.
ricegf@antares:~/dev/202308/10-java-user-interfaces/code_from_slide
> java CheckSum -a MD5 /home/ricegf/logo.png
9df6f40f0a76a36b472ad0c27342d9ca
ricegf@antares:~/dev/202308/10-java-user-interfaces/code_from_slide
```

picocli
(that is, small Command Line Interfaces)

- picocli is a single-file framework for creating Java CLI applications with minimal code
  - Supports flag / option styles for Linux, Windows, and others
  - Auto-generates colorful help and error messages
  - Auto-generates HTML / PDF documentation and Linux man pages
  - Supports tab completion for bash, DOS, and PowerShell
  - Supports creating a single-file executable for distribution
- Simply use annotations similar to @override with parameters to describe your desired user interface

Bare Bones
Utility to Generate File Checksums

This is the skeleton of our application – fields and business logic
(this code will *not* compile – yet!)

Implementing **`Callable`** promises that our application code Is now in method **`call()`**!

```
class CheckSum implements Callable<Integer> {

    private String algorithm = "SHA-256";

    private File file;

    @Override
    public Integer call() throws Exception { // your business logic goes here...
        byte[] fileContents = Files.readAllBytes(file.toPath());
        byte[] digest = MessageDigest.getInstance(algorithm).digest(fileContents);
        System.out.printf("%0" + (digest.length*2) + "x%n", new BigInteger(1, digest));
        return 0;
    }

    public static void main(String[] args) {
        // TBD
    }
}
```

Method **`call()`** is now your application's code
(this was **`printCheckSum()`** in Classic CLI)

picocli

Utility to Generate File Checksums

```java
import picocli.CommandLine;
import picocli.CommandLine.Command;
import picocli.CommandLine.Option;
import picocli.CommandLine.Parameters;

@Command(name = "checksum", mixinStandardHelpOptions = true, version = "checksum 4.0",
         description = "Prints the checksum (SHA-256 by default) of a file to STDOUT.")
class CheckSum implements Callable<Integer> {

    @Option(names = {"-a", "--algorithm"}, description = "MD5, SHA-1, SHA-256, ...")
    private String algorithm = "SHA-256";

    @Parameters(index = "0", description = "The file whose checksum to calculate.")
    private File file;

    @Override
    public Integer call() throws Exception { // your business logic goes here...
        byte[] fileContents = Files.readAllBytes(file.toPath());
        byte[] digest = MessageDigest.getInstance(algorithm).digest(fileContents);
        System.out.printf("%0" + (digest.length*2) + "x%n", new BigInteger(1, digest));
        return 0;
    }

    public static void main(String[] args) {
        System.exit(new CommandLine(new CheckSum()).execute(args));
    }
}
```

Non-picocli imports are omitted

@command describes the *application*

@Option defines a *flag*

@Parameters defines a *positional parameter*

Now main instances picocli.CommandLine and calls execute(args) which calls call().

Because we implemented Callable, picocli.execute "knows" that our application code is in method call()!

This is the *entire application!*

Running Checksum

Nothing special to build – add just *one* file!

Automatic colorful error messages…

… and usage information…

… and version information!

Runs just as great!

```
ricegf@antares:~/dev$ ls CheckSum.java picocli/*.java
CheckSum.java  picocli/CommandLine.java
ricegf@antares:~/dev$ javac CheckSum.java
ricegf@antares:~/dev$ java CheckSum
Missing required parameter: '<file>'
Usage: checksum [-hV] [-a=<algorithm>] <file>
Prints the checksum (SHA-256 by default) of a file to STDOUT.
      <file>      The file whose checksum to calculate.
  -a, --algorithm=<algorithm>
                  MD5, SHA-1, SHA-256, ...
  -h, --help      Show this help message and exit.
  -V, --version   Print version information and exit.
ricegf@antares:~/dev@ java CheckSum --version
checksum 4.0
ricegf@antares:~/dev$ java CheckSum -a SHA-1 CheckSum.java
0515946bfc72224b5f28ccac6fa139aa8aa1dd94
ricegf@antares:~/dev$ java CheckSum CheckSum.java
44a325fa933b5ef7b83138e9721b1227e48cea8d68a9e3f39233f8d94f0661b4
ricegf@antares:~/dev$ sc
```

Thoughts on CheckSum with picocli

- This is 39 (not 61, 59, or 75) lines of Java code – 36% less!
  - All code except main is application-specific
  - Less code means fewer bugs (and UI code tends to be buggy)
- The usage statement is *calculated*
  - If you modify the code, usage message changes to match
- We get a lot of “free” code
  - The --help and --version options
  - Error checking and error message generation
  - Returning error codes to the operating system
- We specify all aspects exactly once
  - You Don’t Repeat Yourself (DRY)

Using picocli

- Obtain a copy
  - From (easy) cse1325-prof/picocli or (latest source)
    https://github.com/remkop/picocli/blob/main/src/main/java/picocli/CommandLine.java
- Check the documentation
  - https://picocli.info/
- Declare your main class as
  `implements Callable<Integer>`
  - And thus write your application code starting in
    `public Integer call() throws Exception {`
- Declare your key fields as `@Option` or `@Parameters`
  - Supports primitives, enums, String / StringBuilder, File, Path, Date (“yyyy-MM-dd”)
  - Also supports ArrayLists for arbitrarily long filename parameters (for example)!
- Include the “boilerplate” (pre-defined) static main method (try/catch optional)
  - `System.exit(new CommandLine(new Classname()).execute(args));`

Your main class’ name goes here!

@Option

- Options are flags with an optional value
  - `ls -l` : the -l is a flag that says “print a long listing”
  - `git --version` : the --version prints git’s version number
  - `head -n 30 X.java` : print the first 30 lines of X.java
- @Option parameters
  - `names = {"-a", "--algorithm"}` : The flags to recognize
  - `description = "MD5, SHA-1"` : Text for the usage message
- A Boolean @Option field is true if flag is specified, false otherwise
  -
```
@Option(names = "-l", description = "long listing")
boolean long;
```
- Otherwise, a value *is* expected and is assigned to the field
  -
```
@Option(names = "-n", description = "number of lines")
int numberOfLines = 10;
```

Default if -n isn’t specified!

```
@Option(names = {"-a", "--algorithm"}, description = "MD5, SHA-1, SHA-256, ...")
private String algorithm = "SHA-256";
```

@Parameters

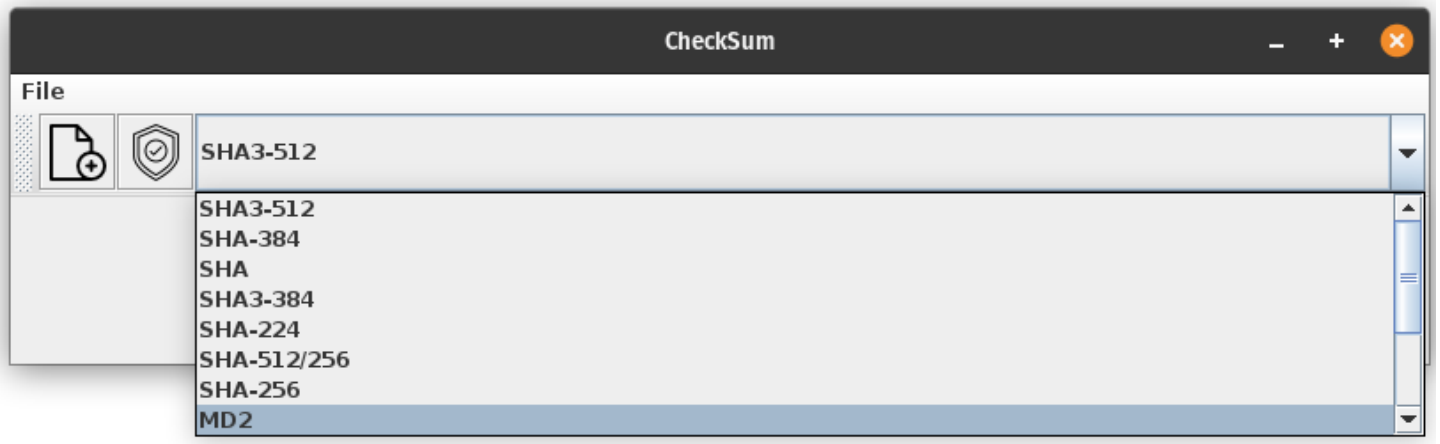
- Parameters are unnamed values with an index (0, 1, 2, ...)
  - **`head -n 30 X.java`** : print the first 30 lines of X.java
  - **`grep -i println *.java`** : search for “println” in all Java files
- @Parameters parameters
  - index = "0" : The position on the command line
  - description = "The search string" : Text for the usage message
- A single valued parameter requires an index
  - ```
    @Parameter(index = "0", description = "search string")
    String searchFor;
    ```
- A multi-valued parameter omits an index and gets all of the remaining parameters
  - ```
    @Parameter(description = "files to search")
    List<File> files;
    ```

This will get you an ArrayList of all files to search!

```
@Parameters(index = "0", description = "The file whose checksum to calculate.")
private File file;
```

Option 5
Graphical User Interface (GUI)

- CSE1325 no long teaches Graphical User Interfaces
  - We formerly used Swing, a part of the standard Java library, to implement GUIs
  - This was replaced by full C++ in summer 2023
- HOWEVER, Swing is analogous to attomenu
  - NOT by coincidence
  - See if you can follow the gist of the code



Swing
Utility to Generate File Checksums

```
public class CheckSumWin extends JFrame {
    public CheckSumWin() {
        super("CheckSum");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(800, 200);

        // /////// ////////////////////////////////////////////////////////////
        // M E N U
        // Add a menu bar to the PAGE_START area of the Border Layout

        JMenuBar menubar = new JMenuBar();

        JMenu     file       = new JMenu("File");
        JMenuItem addFile    = new JMenuItem("Add File");
        JMenuItem csum       = new JMenuItem("Generate Checksums");
        JMenuItem quit       = new JMenuItem("Quit");

        addFile.addActionListener(event -> onAddFileClick());
        csum.addActionListener(event -> onGenerateChecksumsClick());
        quit.addActionListener(event -> System.exit(0));


        file.add(addFile);
        file.add(csum);
        file.add(quit);

        menubar.add(file);
        setJMenuBar(menubar);
```

JFrame is Swing's main window from which we extend our application

Create the menu bar

Add menu items and the method to call when each is selected as with attomenu, except in 2 separate steps.

Add the menu items to the menu.

Add the menu to the menu bar.
Add the menu bar to the main window superclass.

Continued on next page...

Swing
Utility to Generate File Checksums

```
    // ////////////// ////////////////////////////////////////////////////////
    // T O O L B A R
    // Add a toolbar to the PAGE_START region below the menu
    JToolBar toolbar = new JToolBar("Nim Controls");

    JButton addFileButton  = new JButton(new ImageIcon("add_file.png"));
      addFileButton.setToolTipText("Add a file");
      toolbar.add(addFileButton);
      addFileButton.addActionListener(event -> onAddFileClick());

    JButton genCheckSumButton = new JButton(new ImageIcon("gen_checksum.png"));
      genCheckSumButton.setToolTipText("Generate checksums");
      toolbar.add(genCheckSumButton);
      genCheckSumButton.addActionListener(event -> onGenerateChecksumsClick());

    algorithms = new JcomboBox<String>(
        Security.getAlgorithms("MessageDigest").toArray(new String[0]));
      toolbar.add(algorithms);

    getContentPane().add(toolbar, BorderLayout.PAGE_START);


    // ////////////////////////// ////////////////////////////////////////////
    // D I S P L A Y
    // Provide a text entry box to show the checksum table
    display = new JLabel();
    add(display, BorderLayout.CENTER);

    // Make everything in the JFrame visible
    setVisible(true);
}
```

Create the tool bar, which is very similar to the menubar.

(This puts a list of every checksum algorithm in the system in a drop-down called algorithms)

Create the output display area

Run the program

Continued on next page...

Swing
Utility to Generate File Checksums

```
    // Listeners
    protected void onAddFileClick() {
        final JFileChooser fc = new JFileChooser(file);  // Create a file chooser dialog
        int result = fc.showOpenDialog(this);            // Run dialog, return button clicked
        if (result == JFileChooser.APPROVE_OPTION) {     // If OK was clicked,
            file = fc.getSelectedFile();                 // Obtain the selected File object
        }
     }

    protected void onGenerateChecksumsClick() {
        try {
            byte[] fileContents = Files.readAllBytes(file.toPath());
            String algorithm = (String) algorithms.getSelectedItem();
            byte[] digest = MessageDigest.getInstance(algorithm).digest(fileContents);
            display.setText(String.format("<html>%s<br>%0" + (digest.length*2) + "x%n</html>",
                file.getName(), new BigInteger(1, digest)));
        } catch(Exception e) {
            JOptionPane.showMessageDialog(this, "Failed to generate checksum: " + e);
        }
    }

    private File file;
    private JComboBox<String> algorithms;
    private JLabel display;

    public static void main(String[] args) {
        new CheckSumWin();
    }
}
```

Listener methods *react* to button clicks!

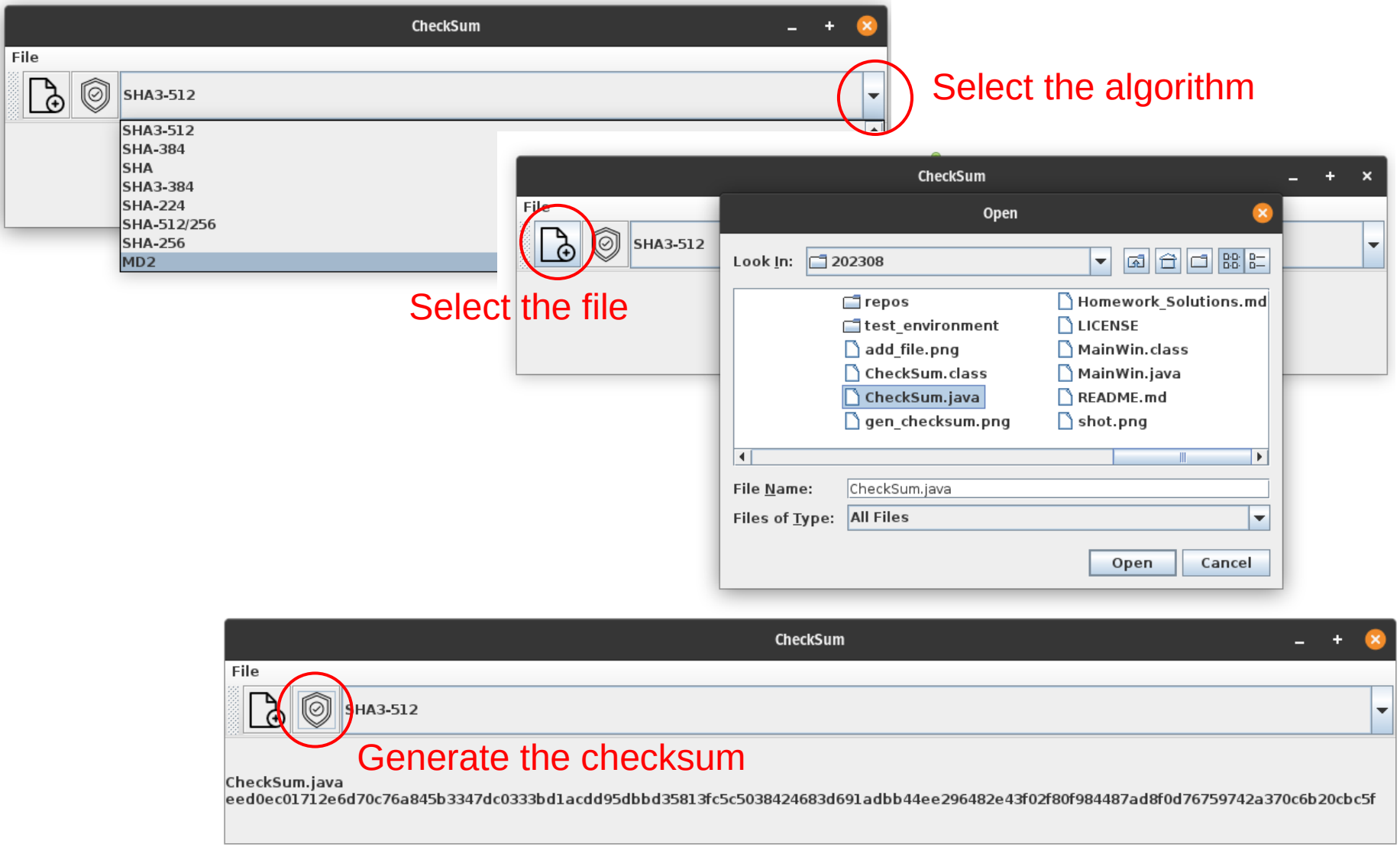
Select the file

Generate the checksum & show in display area

Fields

Main instances and runs the main window

Swing Example



Thoughts on CheckSum with Swing

- This is 75 (not 39 / 61 or 59 / 75) lines of Java code – reasonable for a windowed application!
  - Much of the UI code is in the constructor, including the combo box on the toolbar for selecting the algorithm
  - We use a predefined file chooser dialog
- No usage statement is needed – the interface is *discoverable*
  - We skip the traditional Help > About dialog, though one is traditionally provided to identify the version number and copyright information
  - All error messages are in dialogs – nothing is printed to System.err
  - No error code is returned to the operating system
- We specify all aspects exactly once
  - You Don’t Repeat Yourself (DRY)

User Interface Options
Most of Which are NOT Covered in CSE1325

- Write a custom Menu Driven Interface (MDI)
  - Write your own UI code using System.out and Scanner
- Base your MDI on attomenu
  - Good for menu-driven console applications
  - Less flexible than custom but likely *fewer* lines of code
- Write a custom Command Line Interface (CLI)
  - Write your own UI code using String[] args, System.out, and Scanner
- Base your CLI on picocli
  - Good for command line interface utilities
  - A bit harder to learn, but very flexible and likely *fewer* lines of code
- Base your GUI on Swing
  - It's what users generally expect on the business desktop today