Week 6 - Monday

COMP 3100

Last time

- What did we talk about last time?
- Software quality assurance

Questions?

Maggie Smith 1934 - 2024

User Interaction Design

Interaction design

- **Interaction design** is planning out the **user experience (UX)** for a software product
- It cares about how the product looks and sounds (and, one day, smells?) and how the user gets output and puts input into it
- This field used to get little attention from computer scientists, but it's really important
  - Apple is a great posterchild for showing off the value of UX
  - Even Microsoft, maligned for its user interfaces, has invested lots of money studying how to make windows and icons easier to use
- UX is part of the field of **human computer interaction (HCI)**, which combines ergonomics, physiology, psychology, and graphic design with computer science
- The quality of a user interface is called its **usability**

User interaction design goals

- **Effectiveness:** User can access all the features they need
- **Efficiency:** Users can achieve their goals quickly
- **Safety:** Users and computers aren't harmed
- **Learnability:** Users become proficient quickly
- **Memorability:** Users regain proficiency quickly after time away from the product
- **Enjoyability:** Users experience positive emotions when using the product
- **Beauty:** Users find the product aesthetically pleasing

Blackboard: A case study in terrible UX

- **Effectiveness:** Content isn't easily viewable on phones, and some rows at the bottom or columns on the right can't be clicked on, depending on screen size
- **Efficiency:** It takes something like 7 mouse clicks for me to upload a project feedback form
- **Safety:** My wrists hurts after 7 clicks × 24 students
- **Learnability:** There are features of Blackboard that students never learn about, and issues like the collapsing navigation bar cause perennial problems
- **Memorability:** All the different options for exam timing and turn-in options seem just as confusing each semester
- **Enjoyability:** I want to kill everyone after using Blackboard
- **Beauty:** This one is subjective, and I will admit that it looks better than it did 10 years ago

When to do interaction design

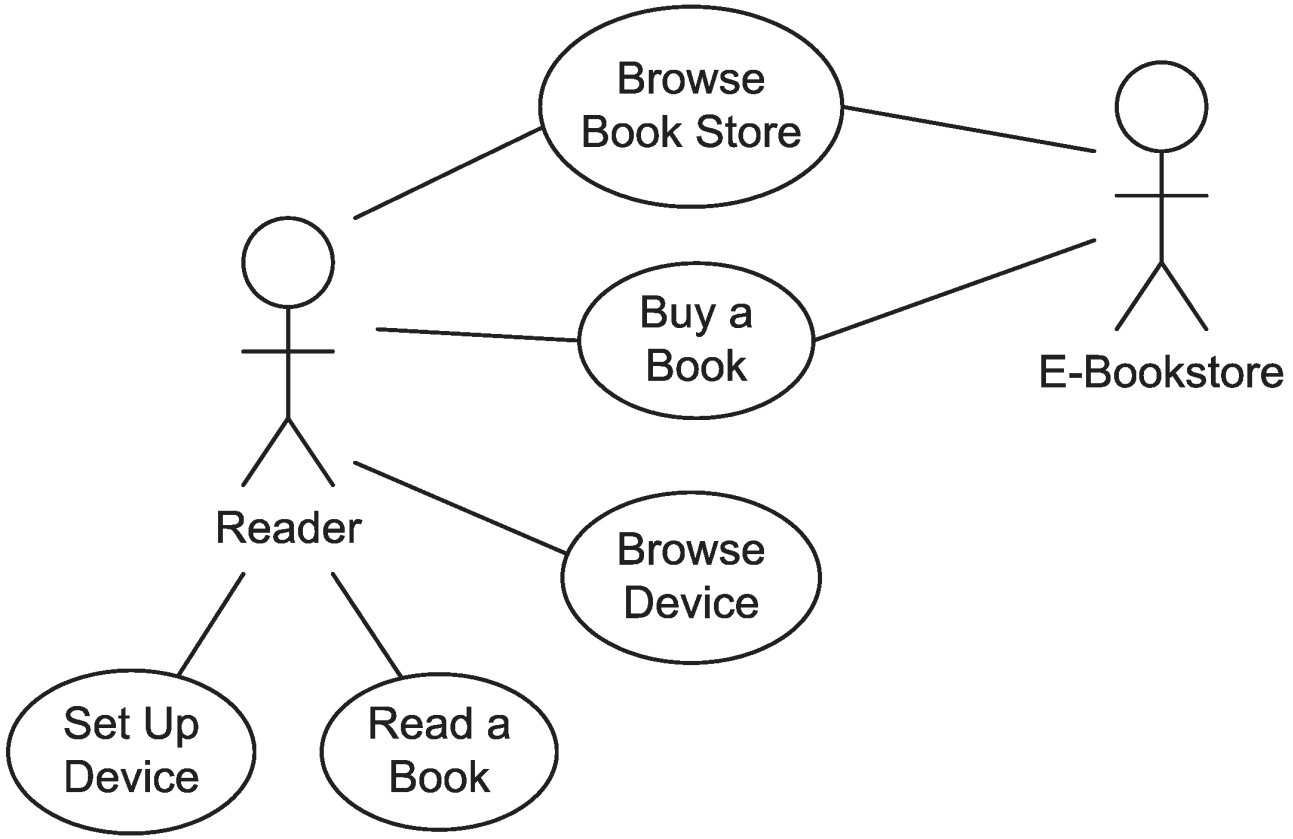
- In the past, people were concerned with the computation and threw in UX as an afterthought
- Now, we realize that UX has a lot to do with the features the product should have
- Interaction design should be a key part of requirements specification
- Some agile people think you should do the interaction design during one sprint for the features that will be implemented in the next sprint

Interaction design models

- Before coding the UX, models are incredibly helpful to plan out how it looks and behaves
- **Static interaction design models** show the audio and visual parts of the product that don't change during execution
- **Dynamic interaction design models** show behavior during execution
- Both are useful

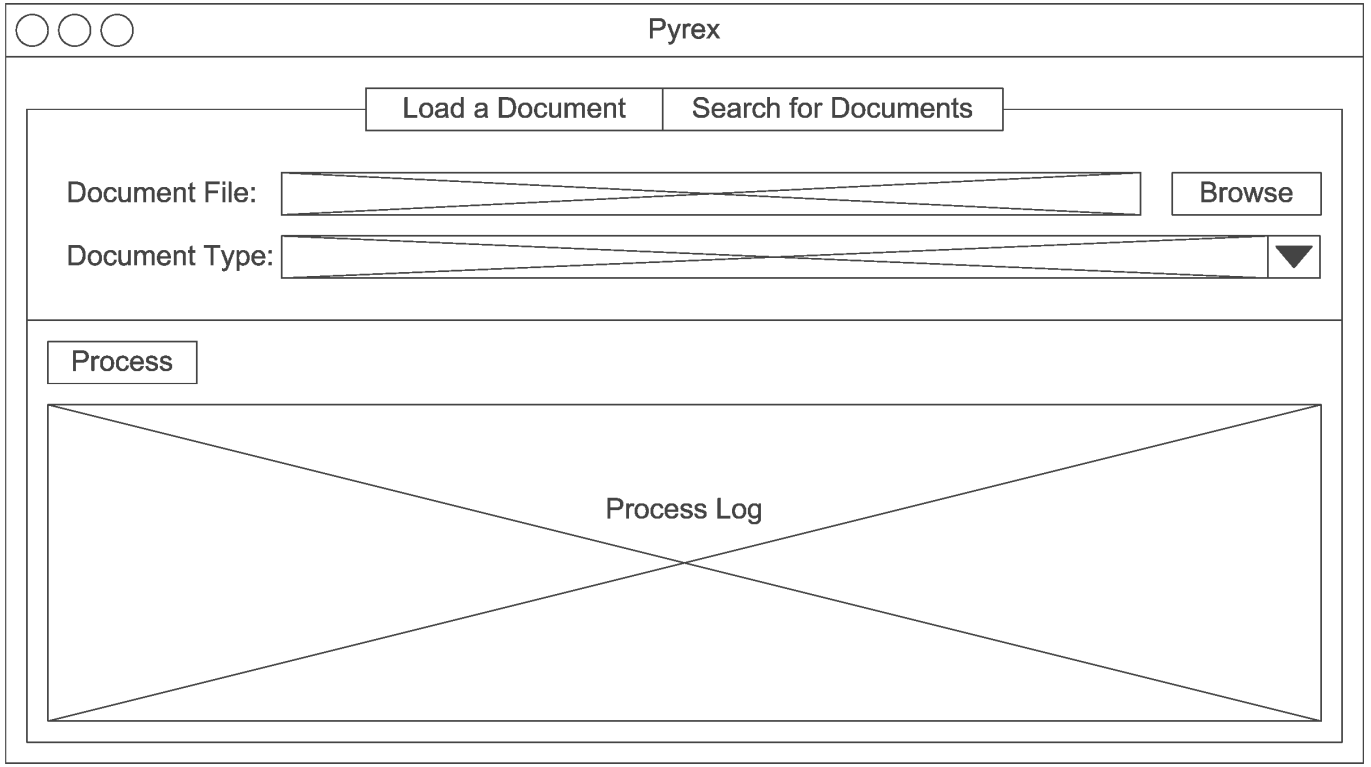
Use case diagrams

- A **use case** is an interaction between a product and its environment
- An **actor** is an agent that interacts with a product
- **Use case diagrams** (which we've seen before) are static interaction design models that represent the actors that interact with use cases



Layout diagrams

- **Screen layout diagrams** and **page layout diagrams** are drawings of a product's visual display
- A **wireframe** is a low-fidelity version that gives a rough layout without a lot of detail
- It's good to start with a wireframe and refine it with more detail later



Use case descriptions

- A use case diagram shows which actors interact with use cases
- However, it doesn't explain what they *do*
- A **use case description** is formatted text that explains the actions that an actor makes
- The use case description is a dynamic interaction design model
- Example template:

Use Case Name	To identify the use case
Actors	The agents participating in the use case
Stakeholders and Needs	What this use case does to meet stakeholder needs
Preconditions	What must be true before this use case begins
Post conditions	What will be true when this use case ends
Trigger	The event that causes this use case to begin
Basic Flow	The steps in a typical successful instance of this use case
Extensions	The steps in alternative instances of this use case due to variations in normal flow or errors

Storyboards

- Another dynamic interaction design model is a **storyboard**
- A storyboard shows different screen layout diagrams connected with arrows
  - The arrows show either the passage of time or responses to events
- Use case descriptions can describe what is happening in interactions, but storyboards show what it looks like
- Like screen layout diagrams, they will usually start simple and then become more detailed

Use case models and requirements

- Use case models can be helpful at the requirements stage
- Even with use case descriptions, they don't give enough information about requirements that are not connected to interactions
  - Like processing the data once it's been received
  - Or non-functional requirements
- They're good at showing interactions but not necessarily how to build the system that makes those interactions work
- In short, use case models are useful but not enough
- Use case models can help elaborate PBIs during sprints

Interaction design processes

- Interaction design should move from abstract models to more concrete, refined ones
- From stakeholder needs, you can make a use case diagram
- These can be elaborated with use case descriptions
- Screen layout diagrams and storyboards help with design and getting feedback
- **Usability testing** uses experiments with test subjects to see if they can interact with the product
  - Testing doesn't have to be on the full, polished product
  - Measurement could be:
    - Asking the subjects questions about their experience
    - Counting their errors
    - Measuring time taken to accomplish tasks

Crash Course on Java Swing

JOptionPane

JOptionPane

- **`JOptionPane`** class provides static methods for:
  - Displaying a message
  - Asking a question
- Although it is possible to create a **`JOptionPane`** object, you almost never do
- Just call the static methods
  - Which means typing a lot of **`JOptionPane.`**

showMessageDialog() example

- To display **"This is a message."** you could call the following:

```
JOptionPane.showMessageDialog(null,
  "This is a message.");
```

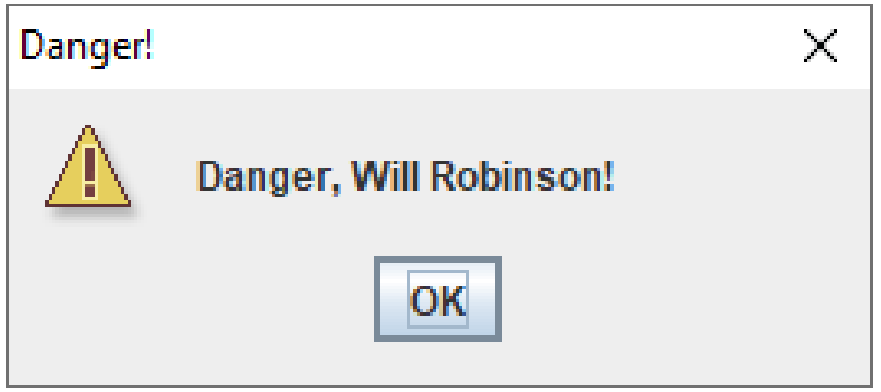
Adding a title

- Most **`JOptionPane`** methods have many overloads
- If you want to put a title on the window, you can pass it in as the third parameter
- But this overloaded method also requires an **`int`** parameter that says what kind of message you want
- To add the title **`"Window Title"`**, you might call the following method:


```
JOptionPane.showMessageDialog(null,
  "This is a message.", "Window Title",
  JOptionPane.PLAIN_MESSAGE);
```

Different icons

- You can choose an icon associated with one of the following constants:
  - **ERROR_MESSAGE**
  - **INFORMATION_MESSAGE**
  - **WARNING_MESSAGE**
  - **QUESTION_MESSAGE**
  - **PLAIN_MESSAGE**




```
JOptionPane.showMessageDialog(null,
   "Danger, Will Robinson!", "Danger!",
   JOptionPane.WARNING_MESSAGE);
```

showConfirmDialog()

- What if you don't just want to display a message?
- You could also display a prompt with yes, no, and cancel buttons, using the following method signature

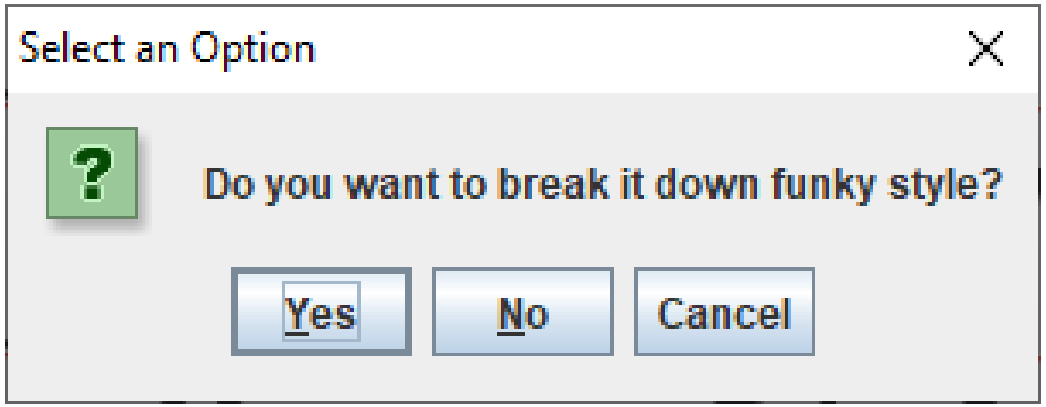
```
int showConfirmDialog(Component parent,
    Object message)
```

- This method will display **message** and return one of the following **int** constants inside **JOptionPane**:
  - **YES_OPTION**
  - **NO_OPTION**
  - **CANCEL_OPTION**

showConfirmDialog() example

```
int answer = JOptionPane.showConfirmDialog(null,
      "Do you want to break it down funky style?");
if (answer == JOptionPane.YES_OPTION) {
      JOptionPane.showMessageDialog(null, "Dope!");
} else {
      JOptionPane.showMessageDialog(null, "Weak!");
}
```

- Hitting the X in the corner is the same as Cancel



showInputDialog()

- A more flexible option for input is **`showInputDialog()`**
- It allows the user to type arbitrary text into a box
- Be careful with the return value
  - If they cancel, it's **`null`**
  - They might put crazy spaces at the beginning and end (use **`trim()`**)
  - Numbers have to be converted from **`String`** values
- Signature:

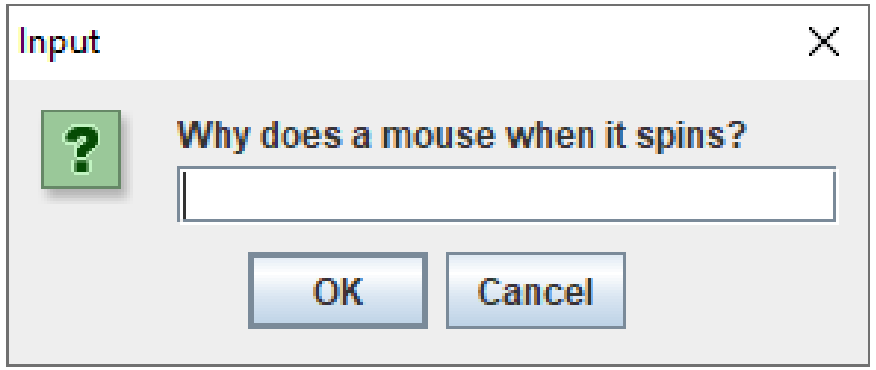
```
String showInputDialog(Component parent,
    Object message)
```

showInputDialog() example

- This input dialog asks a pressing question

```
String answer =
  JOptionPane.showInputDialog(null,
  "Why does a mouse when it spins?");
```

- As with other methods, there are overloaded versions that allow for titles, icons, and other options



JFrame

JFrame

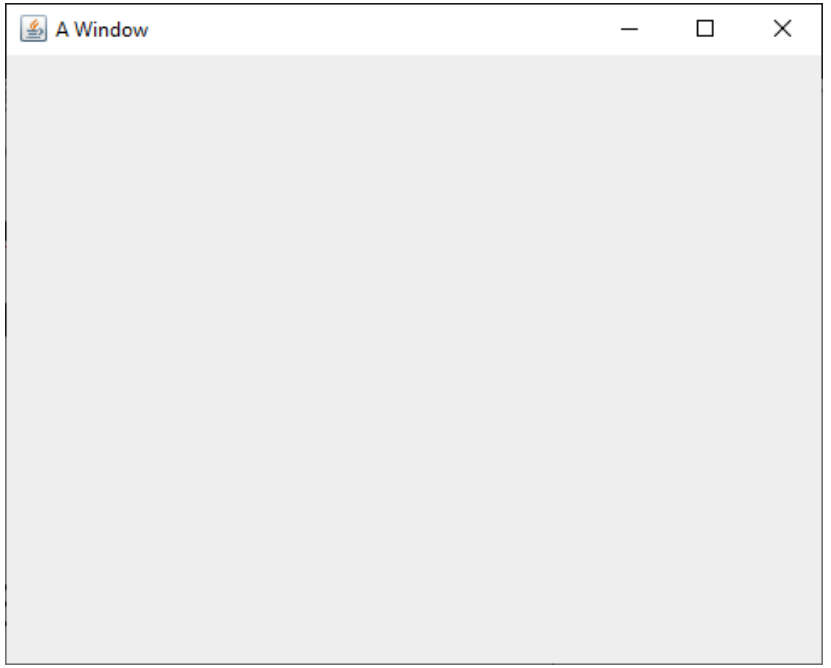
- **JOptionPane** was fine for creating a limited range of dialogs
- If we want to make a whole window, we use **JFrame**
- Java uses the term frame instead of window, probably because of concerns about lawsuits from Microsoft
- But when you hear **JFrame**, think "main window"

Creating or extending

- When designing a **`JFrame`**, there are two meaningful options:
  - Creating a **`JFrame`** object and adding stuff to it inside of some other class
  - Extending **`JFrame`** with your own class, making your class a **`JFrame`** plus more
- It doesn't really matter which one you pick
- To keep things simple, we'll create a **`JFrame`** object instead of extending the **`JFrame`** class

Creating a JFrame

- To create a **JFrame**, we will usually call its constructor that takes a **String**, giving it a title
- Then, we have to make it visible so that we can see it




```
JFrame frame = new JFrame("A Window");
frame.setVisible(true);
```

setSize()

- The code from the previous slide will make a **JFrame** and make it visible
- However, it will probably be so small that you won't even notice it
- To deal with this problem, you should set its size, ideally before you make it visible
  - Its **setSize()** method takes two **int** values: width and height in pixels
- Eventually, once we add widgets to a **JFrame**, we can simply call its **pack()** method, which will make it take up the amount of space it needs to fit everything

```
JFrame frame = new JFrame("A Window");
frame.setSize(500, 400);
frame.setVisible(true);
```

setDefaultCloseOperation()

- Next, you'll notice that closing the window doesn't end the program
  - The little red square on the IntelliJ Console is still clickable, meaning that the program is running
- By default, closing the window by clicking its X only hides the window
- By calling the **`setDefaultCloseOperation()`**, we can make it so that the default operations is dispose (getting rid of the window)

```
JFrame frame = new JFrame("A Window");
frame.setSize(500, 400);
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
frame.setVisible(true);
```

- Many books suggest passing in **`JFrame.EXIT_ON_CLOSE`**, but you **should not!**
- Doing so will kill the rest of your program like **`System.exit()`**

Widgets

Widgets

- **Widget** is a generic term for a wide range of GUI controls
  - Buttons
  - Labels (allowing us to put text or images on a GUI)
  - Text fields
  - Text areas (like text fields but larger)
  - Menus
  - Checkboxes
  - Radio buttons
  - Lists
  - Combo boxes
  - Sliders

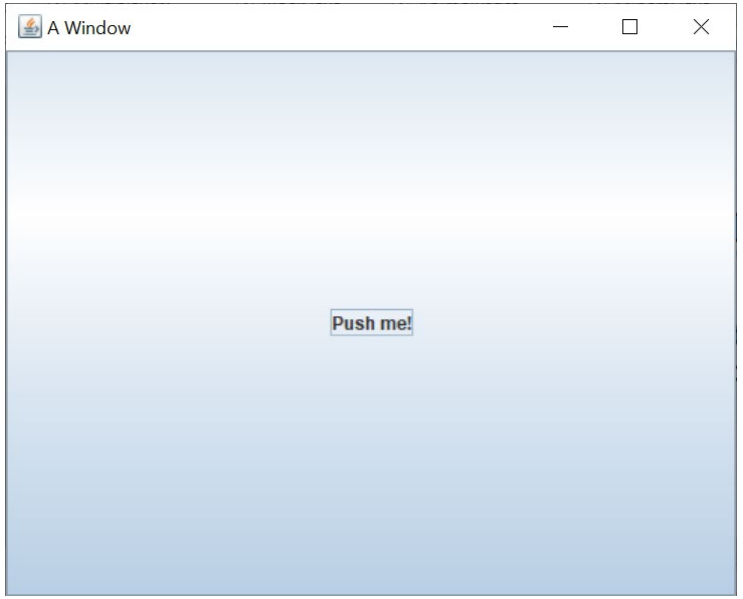
JButton

- A button you can click on is provided by the **JButton** class
- A **JButton** is usually created with text or an image
- Just creating the **JButton** doesn't do anything
- You have to add it to a **JFrame** (or other container) to see it
- Right now, we're just creating the buttons
- Later, we'll learn how to add actions to them

```
JButton button = new JButton("Push me!");
```

Adding a JButton to a JFrame

- Once you've created a **JButton**, you can add it to a **JFrame** by calling the **add()** method on the **JFrame**
- All GUI containers have an **add()** method that allows us to add a widget to it

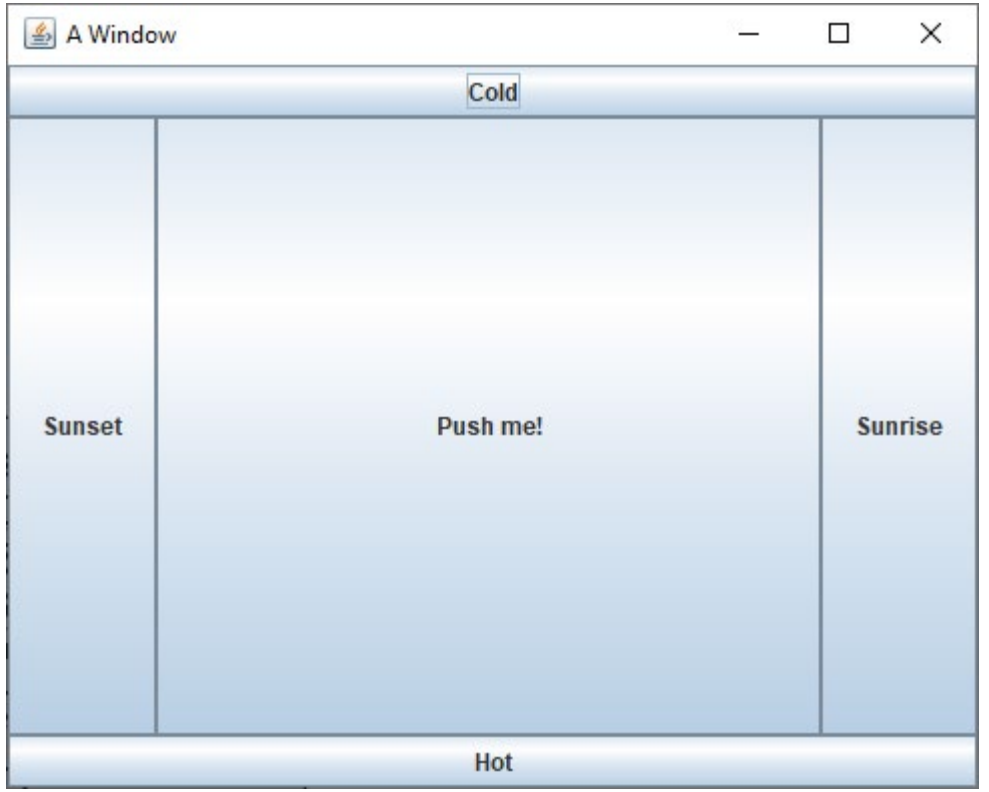



```
JFrame frame = new JFrame("A Window");
frame.setSize(500, 400);
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
JButton button = new JButton("Push me!");
frame.add(button);
frame.setVisible(true);
```

Adding to different parts of a `JFrame`

- By default, a **`JFrame`** uses a layout manager called the **`BorderLayout`** that has five region
- Calling the simplest **`add()`** method adds a widget to the center, which stretches to take up all available space
- You can specify that you're adding to:
  - **`BorderLayout.CENTER`**
  - **`BorderLayout.NORTH`**
  - **`BorderLayout.SOUTH`**
  - **`BorderLayout.EAST`**
  - **`BorderLayout.WEST`**

```
JButton centerButton = new JButton("Push me!");
frame.add(centerButton, BorderLayout.CENTER);
JButton northButton = new JButton("Cold");
frame.add(northButton, BorderLayout.NORTH);
JButton southButton = new JButton("Hot");
frame.add(southButton, BorderLayout.SOUTH);
JButton eastButton = new JButton("Sunrise");
frame.add(eastButton, BorderLayout.EAST);
JButton westButton = new JButton("Sunset");
frame.add(westButton, BorderLayout.WEST);
```



Displaying an icon on a JButton

- You can also make a **JButton** with an image instead of text
- To do so, you create an **ImageIcon** and pass that to the constructor of the **JButton**
- You'll need the path to an image


```
JButton bowieButton = new JButton(new ImageIcon("bowie.jpg"));
frame.add(bowieButton, BorderLayout.CENTER);
```

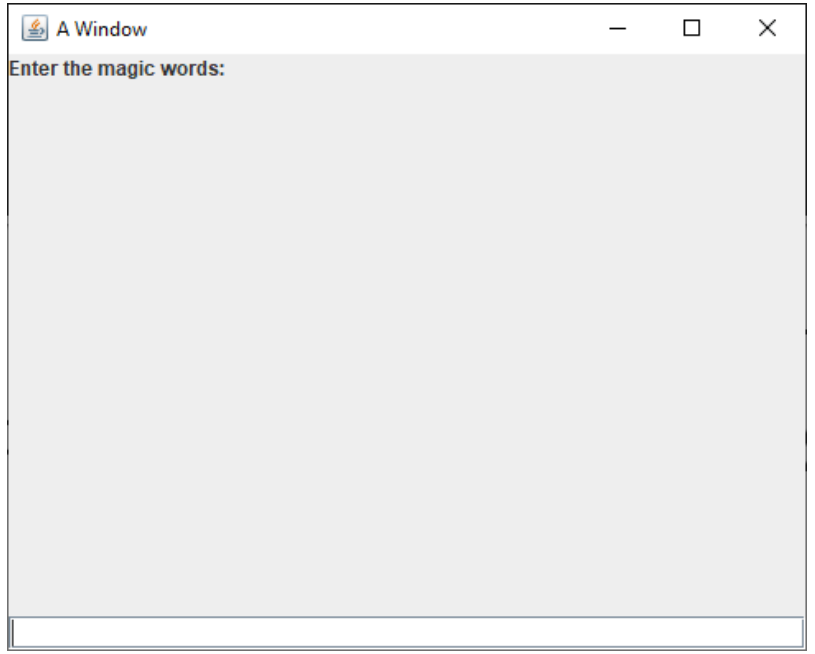
JLabel

- A **JLabel** is like a button you can't click
- Its constructors work just like the **JButton** ones
- It allows you to display text or an image


```
JLabel nameLabel = new JLabel("David Bowie");
JLabel bowieLabel = new JLabel(new ImageIcon("bowie.jpg"));
frame.add(nameLabel, BorderLayout.NORTH);
frame.add(bowieLabel, BorderLayout.CENTER);
```

JTextField

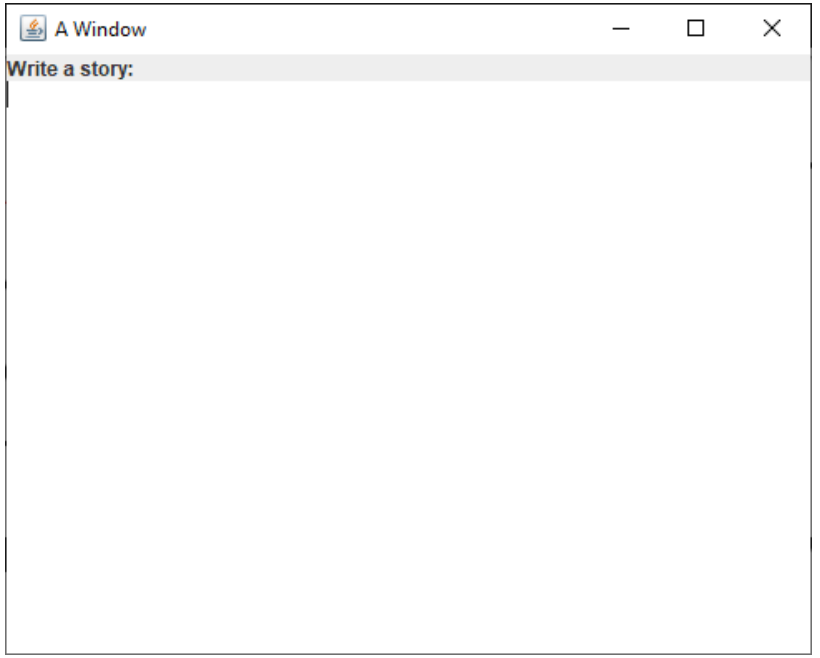
- A **JTextField** allows a user to enter a (short) amount of text
- Usually, you'll need a **JLabel** to tell the person what they should enter
- The example is ugly because the **JLabel** and the **JTextField** don't fill the 500 x 400 **JFrame**




```
JLabel messageLabel = new JLabel("Enter the magic words:");
JTextField magicField = new JTextField();
frame.add(messageLabel, BorderLayout.NORTH);
frame.add(magicField, BorderLayout.SOUTH);
```

JTextArea

- A **JTextField** is for entering small pieces of information
  - Name
  - Address
  - Telephone number
- For larger texts, we can use a **JTextArea**




```
JLabel storyLabel = new JLabel("Write a story:");
JTextArea storyArea = new JTextArea();
frame.add(storyLabel, BorderLayout.NORTH);
frame.add(storyArea, BorderLayout.CENTER);
```

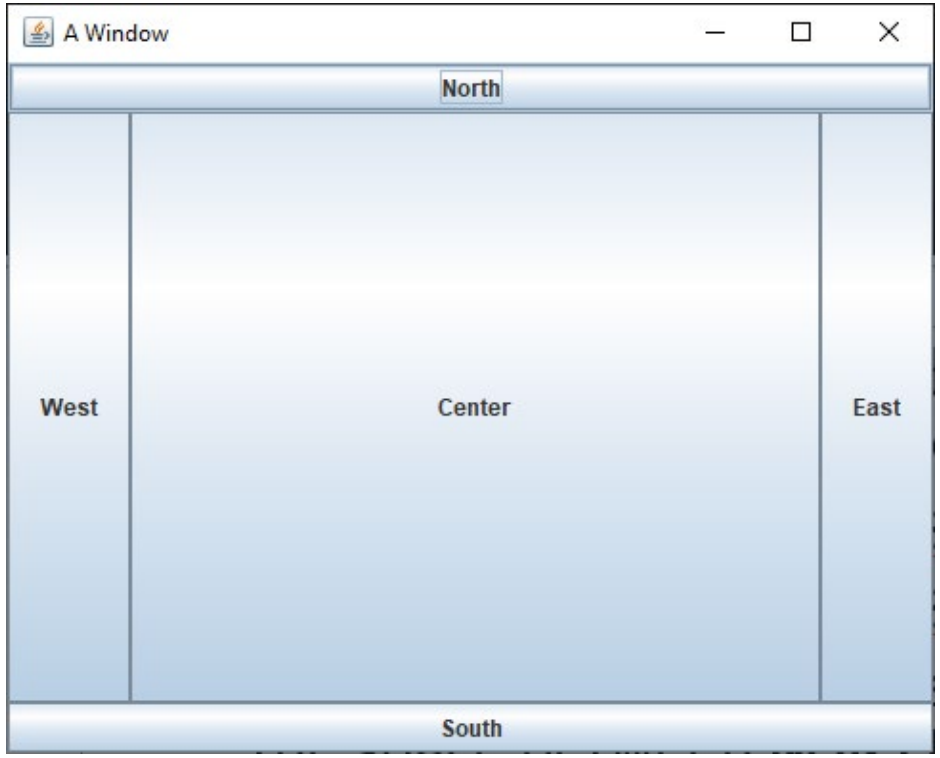
Layout Managers

Layout managers

- When you add a widget to a **`JFrame`** (or to a **`JPanel`**), its layout manager determines how it will be arranged
- There are lots of layout managers, but it's worth mentioning four:
  - **`BorderLayout`**
  - **`GridLayout`**
  - **`FlowLayout`**
  - **`BoxLayout`**
- We won't talk about **`BoxLayout`**, but you should look it up
- **`BoxLayout`** makes it easy to arrange widgets in a horizontal or vertical line, with different amount of spacing between widgets

BorderLayout

- **`BorderLayout`** is the default layout for **`JFrame`**
- When you add widgets, you can specify the location as one of five regions:
  - **`BorderLayout.NORTH`** stretches the width of the container on the top
  - **`BorderLayout.SOUTH`** stretches the width of the container on the bottom
  - **`BorderLayout.EAST`** sits on the right of the container, stretching to fill all the space between **`NORTH`** and **`SOUTH`**
  - **`BorderLayout.WEST`** sits on the left of the container, stretching to fill all the space between **`NORTH`** and **`SOUTH`**
  - **`BorderLayout.CENTER`** sits in the middle of the container and stretches to fill all available space
- If you don't specify where you're adding a widget, it adds to **`CENTER`**
- If you add more than one widget to a region, the new one **replaces** the old
- Unused regions disappear



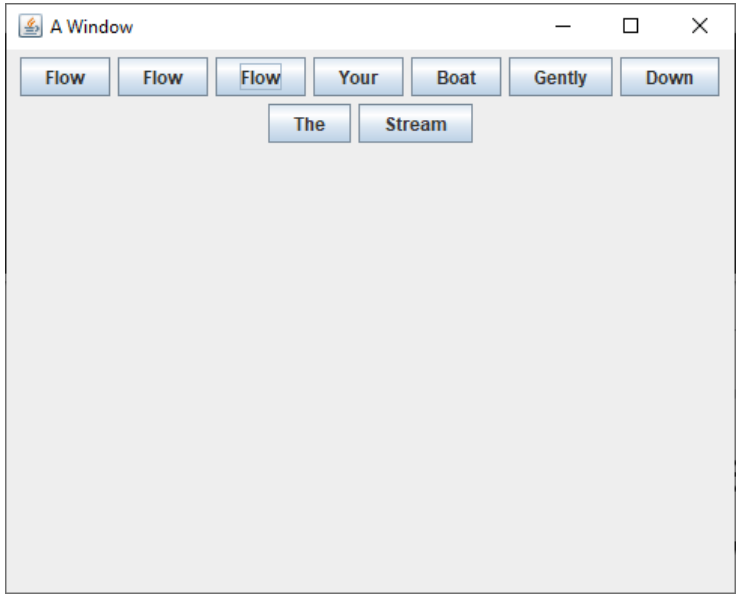
GridLayout

- **GridLayout** allows you to create a grid with a specific number of rows and columns
- All the cells in the grid are the same size
- As you add widgets, they fill each row


```
frame.setLayout(new GridLayout(4, 5));
for (int row = 0; row < 4; ++row) {
  for (int column = 0; column < 5; ++column) {
     frame.add(new JButton("" + (row * 5 + column + 1)));
  }
}
```

FlowLayout

- **FlowLayout** is the default layout manager for **JPanel**
- Widgets are arranged in centered rows in **FlowLayout**
- If you keep adding widgets to a **FlowLayout**, they'll fill the current row until there's no more room
- Then, they'll flow onto the next row
- It's ugly but easy to use



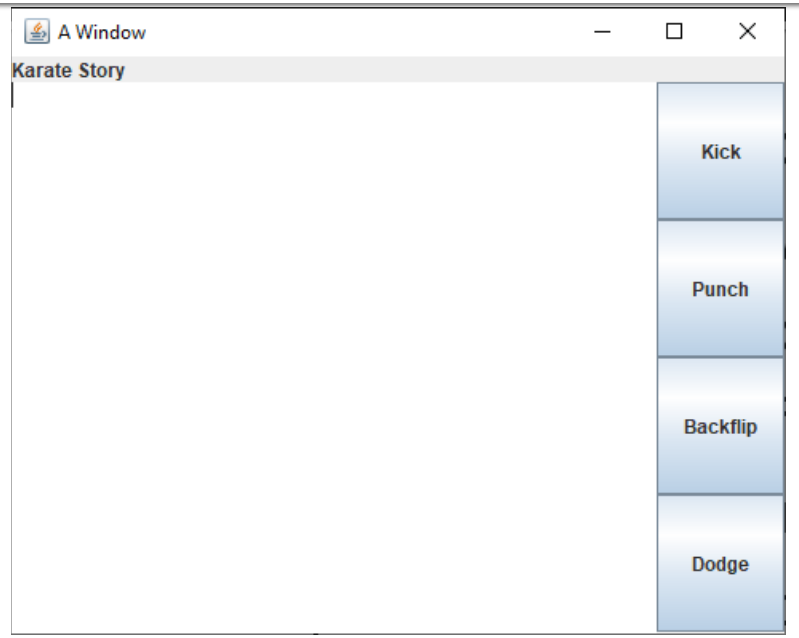

```
frame.setLayout(new FlowLayout());
frame.add(new JButton("Flow"));
frame.add(new JButton("Flow"));
frame.add(new JButton("Flow"));
frame.add(new JButton("Your"));
frame.add(new JButton("Boat"));
frame.add(new JButton("Gently"));
frame.add(new JButton("Down"));
frame.add(new JButton("The"));
frame.add(new JButton("Stream"));
```

JPanel

- A **JPanel** is an invisible container that:
  - Acts like a widget in that you can add it to a **JFrame** (or another **JPanel**)
  - Can hold other widgets
  - Can have its layout customized with a layout manager
- What if you have a **BorderLayout** and you want the **EAST** region to contain widgets arranged with a **GridLayout**?
- Easy: you create a **JPanel**, set its layout manager to **GridLayout**, add it to the **EAST** region, then add widgets to the **JPanel**

Complicated layouts

- For complicated layouts
  - Sketch out what you want it to look like
  - Use **`BorderLayout`**s to give components a spatial relationship
  - Nest **`JPanel`**s inside of **`JPanel`**s (inside of **`JPanel`**s…) if you need to
  - Use **`GridLayout`**s whenever you want to have a grid
  - Be patient: it's hard to get it right the first time




```
JPanel buttonPanel = new JPanel(new GridLayout(4,1));
buttonPanel.add(new JButton("Kick"));
buttonPanel.add(new JButton("Punch"));
buttonPanel.add(new JButton("Backflip"));
buttonPanel.add(new JButton("Dodge"));
frame.add(buttonPanel, BorderLayout.EAST);
frame.add(new JLabel("Karate Story"), BorderLayout.NORTH);
frame.add(new JTextArea(), BorderLayout.CENTER);
```

Action Listeners

Making buttons do things

- We have added **`JButton`**s to **`JFrame`**s, but those buttons don't do anything
- When clicked, a **`JButton`** fires an event
- We need to add an action listener to do something when that event happens
- A CLI program runs through loops, calls methods, and makes decisions until it runs out of stuff to do
- GUIs usually have this **event-based** programming model
- They sit there, waiting for events to cause methods to get called

ActionListener interface

- What can listen for a **JButton** to click?
- Any object that implements **ActionListener**
- **ActionListener** is an interface like any other with a single abstract method in it:

```
void actionPerformed(ActionEvent e);
```

- We need to write a class with such a method
- We will rarely need to worry about the **ActionEvent** object
- But it does have a **getSource()** method that will give us the **Object** (often a **JButton**) that fired the event

Adding an action listener

- Using an anonymous inner class, we can make an **`ActionListener`** object right when we need it, for a button

```
JButton button = new JButton("Push me!");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setText("Ouch!"); // arbitrary code
    }
}); // ugly: parenthesis for end of method call
```

- It's ugly, but it works

Things you might do in an action listener

- Call arbitrary methods
- **`setText()`** sets the text on many widgets
- **`getText()`** gets the text from widgets so you can do something with it
- Both **`setText()`** and **`getText()`** apply to:
  - **`JButton`**
  - **`JLabel`**
  - **`JTextField`**
  - **`JTextArea`**
- **`setIcon()`** sets the icon on many widgets
  - **`JButton`**
  - **`JLabel`**
- **`setEnabled()`** can be used to enable and disable buttons

Java 8 style

- Before Java 8, we only had two choices:
  - Make a whole class that implements **`ActionListener`** and might have to do different actions based on which button fired the event
  - Make a separate anonymous inner class for every single button, each doing the action for that button
- Java 8 adds something called lambdas which actually make anonymous inner classes too, but the syntax is much nicer
- Java 8 style:

```
JButton button = new JButton("Push me!");
button.addActionListener(e -> button.setText("Ouch!"));
```

More on Java 8 style

- An interface with only a single method in it (like **`ActionListener`**) is called a **functional interface**
- Java 8 lets us instantiate functional interface by filling out the method:

```
(Type1 arg1, Type2 arg2, …) -> { /* method body */  }
```

- But if it's possible for the compiler to infer the argument types, they don't have to be written
- If you only have a single argument, you don't need parentheses
- And if you only have a single line in your method body, you don't need braces
- Multi-line example:

```
JButton button = new JButton("Push me!");
button.addActionListener(e -> {
    button.setText("Ouch!");
    button.setEnabled(false);
});
```

Upcoming

Next time…

- Visual design principles
- Software engineering design

Reminders

- Read Chapter 7: Software Engineering Design for Wednesday
- Keep working on the draft of Project 2
  - Due Friday of next week