**Lesson 1 – Input & Output**

* Input and Output is accomplished in 2 ways
  + Using **Scanner** class
  + System.out (for output)
* User Input 3 step process
  + Create a **Scanner** Object
  + Check Input for Errors
  + Input Data
* **Scanner**
  + Object that has methods that allow us to get data from the user
  + **Sample Methods**
    - **String nextLine()**
      * Returns all text in the current line
    - **String next()**
      * Returns the next token (skipping any whitespace)
    - **Int nextInt(), double nextDouble(), etc**
      * Returns next Token as the specified type (skipping any whitespace)
* Must check the user input to ensure the data is the right type before accessing it
* Convenient to make a class full of static methods that do error checking for you whenever you get user input
* Use the **print()** or **println()** method to output data
  + **print()** method if you wish to stay on the same line
  + **println()** method if you want to go to the next line
* **Escape Sequences =** Ways to format our output (they start with a backslash “\”)
* **printf()** method is a method of formatting the output
  + It contains this information: %[flags][width][.precision] conversion-character **NOTE: Everything in the square brackets is optional and the 2 beginning and end items are required**
    - **%** = Represents a new format item
      * You can have different format items for each object to be outputted
    - **[flags]** = Piece of data that can or cannot be outputted
      * “-“ means left-justify (default is to right-justify)
      * “+” outputs a plus or minus sign for a numerical value
      * “ “ (space) outputs a minus sign if the number is negative or a space if it is positive
    - **[width]** = Specifies the number of characters that must be in the output
      * Unused characters will be displayed as spaces
    - **[.precision]** = Specifies the number of decimal places in a floating-point value
    - **conversion-character** = Tells us what type of data we are trying to output
      * **d** = Decimal Integer (byte, short, int, or long)
      * **f** = Floating-point number (float or double)
      * **c** = Character (Capital C will uppercase the letter)
      * **s** = String (Capital S will uppercase all the letters)

**Lesson 2 – Swing: Layout Managers**

* To make our GUIs visually appealing our components must be arranged in a logical order
  + Done using **Layout Manager**
  + **Types of Layouts given to us by Swing**
    - **Flow Layout (Default)**
    - **Grid Layout**
    - **Border Layout**
    - **Box Layout**
    - **GridBag Layout**
* To use a Layout Manager, we must do 2 things
  + **Instantiate** a LayoutManager Object
    - LayoutIChose variableName = new LayoutIChose();
  + Set the layout to the content pane
    - myPanel.setLayout(variableName);
  + You can now add your components to the JPanel
* **Flow Layout**
  + This is the default layout strategy
  + This layout adds components in a line until the end of the row (left -> right)
    - this.add(this.component);
  + Length of a row depends on width of JPanel and preferred size of components
  + Individual rows are centered
    - Can be changed if desired
* **Grid Layout**
  + All components are laid out in a grid of x rows and y columns
  + X and Y are determined on creation of the layout object
  + Create a Grid Layout Object by:
    - *GridLayout layout = new GridLayout(int x, int y);*
  + Components are added left to right, top to bottom
  + All components are the same size (ignoring preferred size)
* **Border Layout (Usually most useful)**
  + Adds up to 5 components
  + Divides into 5 different areas
    - North, West, South, East, and Center
    - North and South = Entire width
      * Height is determined by the preferred height
    - West and East cover the remaining height
      * The width is determined by the preferred size
    - Center expands to fill any remaining space
  + Positions without any components do not take any space
    - E.g. if EAST does not take up any space then center will fill it
  + Components are added by specifying their desired location
    - *panel.add(component, BorderLayout.NORTH);*
* **Box Layout**
  + Allows us to arrange components in either
    - Horizontal Row
    - Vertical Row
  + *BoxLayout layout = new BoxLayout(Container target, int axis);*
    - *Where axis is either X\_AXIS or Y\_AXIS*
  + Component size in the opposite orientation matches the largest preferred size
  + Components are added left to right or top to bottom (depending on axis)
* **GridBag Layout**
  + The most flexible and most complex Layout Manager
  + Places components in a grid from left to right and top to bottom
  + Each cell in the grid can vary in size
  + Each component can take up more than 1 cell
  + Requires the use of GridBagConstraints objects
* **Creating Complex Layouts**
  + Complexity can be added by **adding panels into panels**
    - EXAMPLE:
      * Place a GridLayout in the East position of a BorderLayout

**Lesson 3 – Files**

* When using files, we must import the java.io package
* We also need to import the java.util package (to get the Scanner package)
* When working with files we must follow a **3-step process**
  + **Open the File**
  + **Input / Output data**
  + **Close the file**
* **Open the File**
  + Create a File Object
    - *File file = new File (String filename);*
  + Then pass the File Object into the Scanner’s constructor
    - *Scanner input = new Scanner(file);*
  + The program will crash if the file is not found
    - Prevent a crash by using a Try-Catch statement and throw a FileNotFoundException
  + To ensure a file exists the instantiation could be placed in a loop that runs until the file is not found
    - We can give the user more information on why an error is occurring using **File Methods**
      * E.g isDirectory(), exists(), canRead(), etc.
* **Input (Line by Line) using Scanner**
  + **Uses the following Scanner Methods**
    - Boolean hasNextLine()
      * Determines if there is another line in the file
    - String nextLine()
      * Returns all remaining text in the current line of the file and sets the cursor to the beginning of the next line in the file
    - String next()
      * Returns the next Token
        + Skipping any whitespace
    - int nextInt(), double nextDouble(), etc
      * Returns the next Token as the specified type
        + Skipping any whitespace
        + Throws InputMismatchException if it is not the right type
  + **Delimiters (Helps prevent errors / bugs)**
    - Characters that separate one Token from the next
    - The Default Delimiters are whitespace characters
      * Space, tab, endOfLine
    - If a data item has a space as part of the data, it can mess up the input algorithm
    - Instead, data items can be separated by a different delimiter
      * Like “:”
    - A Delimiter can be setup with the following code
      * *ScannerObject.useDelimiter (String delimiter);*
* **Output**
  + To output to a file we need to create a PrintWriter Object that takes the filename as its parameter
  + Must close a file when we are done outputting data
    - Files take up a lot of resources so its important to close them
    - Data will not be saved in the file we are outputting to if it is not closed

**Lesson 4 – Swing: Files / Images**

* **JFileChooser**
  + Dialogue box used to allow the user to access files
  + **Two methods** determine whether the dialogue will be used to save or load files
    - *int showSaveDialogue(Component parent);*
    - *int showOpenDialogue(Component parent);*
  + **int showSaveDialogue(Component parent);**
    - **parent**
      * Parent Component that the JFileChooser is attached to
        + The dialogue will be opened on top of this
        + If this is a Frame the JFileChooser will react to commands given to the frame (minimize, close, etc)
        + Can be set to null which will make the JFileChooser independent
    - **int** 
      * Returns an integer representing the option selected
        + APPROVE\_OPTION (User chose a file)
        + CANCEL\_OPTION (User cancelled dialogue)
        + ERROR\_OPTION (Error occurred)
    - If the file was chosen you can get the File Object with getSelectedFile()
      * Returns the selected File
        + You can now perform File Operations
* **Images**
  + One way to work with images is to create an **ImageComponent that extends JComponent**
    - The image is loaded from a jpg, gif, or png file and stored as an ImageIcon Object
    - The actual picture can be accessed using the getImage() Method
    - It is displayed by overriding the paintComponent() Method and calling the drawImage() Method from the Graphics Class
  + **Animation using Images**
    - Another way to display animations is to create an array of a sequence of Image Files
      * + Use a Sleep() Method to slow it down
      * The image displayed is switched between array elements
      * The image swapping can be performed in a separate Thread

**Lesson 5 – Data Types**

* Java includes 8 **Primitive Types**
  + Byte
  + Short
  + Int
  + Long
  + Float
  + Double
  + Char
  + Boolean
* **String is not a Primitive Data Type, it is a Class**
* **Primitive Data Types**
  + Used to help speed up the execution of the program
    - Use less memory
    - Have no Methods
    - Cannot be executed
    - Cannot be extended
  + Can get around this by using a **Wrapper Class**
    - **Wrapper Class** = Class that provides Methods for Primitive Data Types
      * **Example:** type = double, wrapper class = Double
* **Integer Data Types**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Lower Limit** | **Upper Limit** | **Precision** |
| Byte | -128 | 127 | Exact |
| Short | -32 768 | 32 767 | Exact |
| Int | -2 147 483 648 | 2 147 483 647 | Exact |
| Long | -9 223 372 036 854 775 808 | 9 223 372 036 854 775 807 | Exact |

* + Bigger Range = More memory it can store
* **Float Types**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Lower Magnitude** | **Upper Magnitude** | **Precision** |
| Float | + or - 1.4023 E-45 | + or - 3.4028 E+38 | 7 significant digits |
| Double | + or – 4.9406 E-324 | + or – 1.7969 E+308 | 16 significant digits |

* + Because floats and doubles are not precise problems can arise with precision
* Data is automatically converted from Integer Types to a Float Type
* Converting from Float Types to integer Types it requires a **cast**
  + **Casting** loses precision as it does not round the number it just takes everything before the decimal
  + To round a Float Type, we must use methods from the Math Class
  + Casting to a Type that stores less can cause the value to lose precision
* **Boolean Type**
  + Can be converted to an int (1 or 0)
  + Can be converted from an int
* **Char Type**
  + Stores a single Unicode Character
  + Written with single quotes
  + Special characters can be stored using escape sequences
  + Can be converted from