**Lesson 6 – Programming Process**

* 5 Stages of the Programming Process
  + **Defining the Problem**
  + **Planning the Solution**
  + **Coding the Program**
  + **Testing the Program**
  + **Documenting the Program**
* **Defining the Problem**
  + Figure out how the layout of the program
    - What classes are required?
      * What attributes?
      * What services?
    - How are the classes related?
      * Inheritance?
      * Related?
  + The layout is done using a Class Diagrams

|  |
| --- |
| **ClassName goes here** |
| **Attributes**  e.g.  int street  Direction aDirection |
| **Services**  e.g.  Constructor  Query methods  Command methods |

* + A relationship between 2 classes is shown with an arrow
    - Fully enclosed arrow = Inheritance (“is a”)
      * One class inherits another class
    - Open ended arrow = Related (“has a”)
      * One class has an object of another class
    - Also called **Composition**
    - The arrow goes from the **Client** to the **Server**
  + Class Diagrams also show **multiplicity**
  + **Multiplicity =** Notifying how many objects a class contains of the other
    - Often only used for Related Classes
* **Planning the Solution**
  + Every service needs an algorithm outlining how it will work
    - Two options for creating an algorithm
      * Flowchart
        + Visual representation of the flow of the program

Shapes and their representation

Oval = Start / End

Arrow = Program Flow

Rectangle = Process or some sort of command

Parallelogram = I/O interaction with user

Diamond = Decision

Circle with a number = Connector

Flag = Off-page Connector

* + - * Pseudocode
        + Logic structures but no programming syntax

No language specific terminology

* **Coding the Program**
  + Translate the algorithm into a formal programming language
* **Testing the Program**
  + Ways to test our program
    - Use the compiler to detect syntax errors
      * Translate source code to binary code
    - Debugging
      * Run the program using test cases
      * Detect logic errors and rectify them
        + Logic errors = Errors in the way the program works
* **Documenting the Program** -> Performed throughout development
  + Class Diagrams
  + Algorithms
  + Javadocs
  + Testing procedures
  + Help Menus
  + User’s Manual

**Lesson 7 – Stepwise Refinement**

* **Algorithm** = Finite set of step-by-step instructions that specifies a process of moving from the initial situation to the final situation
  + An **Algorithm** is basically a solution spelled out step by step
  + Qualities of Good Algorithms
    - Correct
    - Easy to Read and Understand
      * Each method is composed of few easily understood statements
      * Each method has a single well-defined purpose
      * Each method can be understood by examining its statements
    - Easy to debug
    - Easy to modify to solve variations of the original task
    - Efficient
* **Stepwise Refinement** = Method for program design (Method of constructing algorithms)
  + A design method where you decompose a complex problem into small sub-problems.
  + **Steps:**
    - Start with main method
    - Identify/write any required services
      * Gloss over complex sections of the problem
    - Identify “helper” services
      * Methods that help perform the main task
      * Can be protected methods
    - Repeat
* Stepwise Refinement Advantages
  + Easy to understand
    - Since it is broken into specific sections
  + Easy to debug
    - Since it is broken into specific sections
  + Easy to modify (scalability)
    - You can go in and fix/update any helper service
* Always look to improve your program
  + E.g. make it more efficient / scalable
* **Stub** = Indicate a section of code that requires refinement
  + Stubs should be given an empty header so the class can be compiled
    - Code is then created for the empty stubs
  + Stubs are useful for 3 reasons
    - Stubs serve as placeholders for work that must still be completed
    - Stubs allow for the program to be compiled even though it is not finished
    - A compiled program can allow for early testing to be performed that validates our ideas and helps us uncover bugs
      * Can verify that an initial situation runs perfectly
* Use similar terminology throughout your program to make it easy to understand
* **Primitives** = Methods we already have from an inherited class
* **Top-Down Design** = Starting at the top of the program and working down
  + Similar to the early design process making it sometimes easier to develop
* **Bottom-Up Design** = Starting at the bottom of the program and working up
  + Allows for method testing to be done a lot earlier

**Lesson 8 – Threads**

* All programs operate sequentially (line by line)
  + Not very efficient
* **Thread** = Separate flow of control (new flow of control)
  + Sequence of statements that run on their own (independent of the rest of the program)
    - Examples of where you can see this:
      * Main Method = separate thread
        + Once executed, code is running line by line
      * Printing a piece of paper = separate thread
        + You can print and do other things on your computer
      * Animations
        + Occur independently of other things in the program
  + When 1 or more thread exists the computer much switch between them
    - Statements within each thread are still executed in order
    - Statements from other threads may execute between any 2 statements from the same thread
    - Processor’s execution is fast enough that it appears all steps are being executed simultaneously
  + To use a thread, you must do the following
    - A constructor for the thread object
      * Object that needs to be run past as a parameter
        + Making a new thread to run the object
        + Use the start() method to invoke the thread

This invokes the run() method

run() method has the instructions the thread will execute

Any class using a thread has to have a run() method in it

* + - A command to start the thread
  + Summary of things needed to start a thread
    - Instructions for the thread in the run() method
    - Implement Runnable interface to tell Java your class is setup to run a thread
    - Start the thread
  + To make a run() method you must use an **interface**
    - **Interface** = list of all methods that will be provided to a class that implements it
      * Only state the method signature in the body
* **Warnings -> Side Effects of Threads**
  + Interactions between the threads are very limited
    - Example: 1 Robot needing to wait until the others are finished
  + Change in program state between commands in a particular thread
    - If the state changes in 1 point in time the program may crash

**Lesson 8B – GUI Helper Methods**

* **Access Modifiers** = Control which clients can invoke a method
  + Also known as the **Scope** of a method
* **Protected Methods** only allow the subclass to use/override methods in the superclass

**Lesson 9 – Selection Structures**

* There are 3 basic programming structures
  + **Sequence** = Normal flow where code is executed line by line in the order it appears
  + **Selection** = Where a decision is made whether to branch to another section of code based on the comparison of two values
  + **Repetition** = Where a black of code is repeated numerous times
* **Selection Structure =** A point in the program where 2 values are compared resulting in the flow of execution to be directed to other sections of code
  + **Comparisons** result in a Boolean TRUE or FALSE answer
  + Types of Selection Structure
    - **If Statement** = Decide between 2 or more true or false options (use when options are related to each other)
      * **If Statements execute the code once or not at all**
      * **Logical comparison** is in the first line of the if statement surrounded in brackets
        + Logical Comparison usually contains a **relational expression**

Can also be a method that returns a Boolean value

* + - **Switch/Case statement** = Choose 1 of several options (menu type scenario)
      * Selection is made by testing a particular integer against several integer constants.
      * If a match is found the sequence of statements for that match are executed.
      * The execution continues until a **break** statement is executed which will cause the program to exit that entire block of code
      * **Default** = If none of the Switch statements are true this will execute
      * You can also use characters or Strings as Cases in Switch Statements

**Lesson 10 – Repetition**

* **Repetition is a type of programming structure**
* **Repetition Structure** = A point where a block of code is repeated 0 or more times
  + An ending condition is set, and the loop continues to execute a block of code until the condition is met.
* **For Loop (Counted Loop)** = Repeats the code a set number of times
  + **Sections:**
    - **Section 1 =** for
    - **Section 2** = Control Statements
      * **Initialization** = Assigns an initial value to the loop counter
      * **Test Condition** = Statements will only run if it evaluates to true
      * **Incrementation** = Updates loop index by incrementing or decrementing the loop counter
    - **Section 3** = Body of Repetition (code being repeated)
* **While Loop (Conditional Loop)** = Repeats the code indefinitely until a condition is met
  + **Sections:**
    - **Section 1 =** while
    - **Section 2 =** Control Statements
    - **Section 3 =** Body of Repetition
* **Do While Loop** = Code runs at least once as the condition is tested at the end
  + Rarely Used
* **Nested Loop** = 1 or more loops within a loop

**Lesson 11 – Swing: Scaling Images**

* Scaling Swing Components makes the code a lot of scalable
  + This is because its size can easily be changed, and the values are not hard coded in and can be changed dynamically
* We do this by basing our component’s size on the size of the component
  + We do this with the **getWidth() and getHeight() methods** 
    - This lets us draw our component on a grid
  + We use the Graphics2D class to make the scaling less tedious
    - We only pass in the numerators to each fraction

**Lesson 12 – Instance Variables**

* **Instance Variables** 
  + Instance of an Object **(Attributes)**
  + **Scope** = Extends through their entire class
    - Declared outside of all methods but within the class
  + **Lifetime** = Lifetime of the Object
* Each Object has its own set of Instance Variables
* Variable Declaration sets aside space in memory to store a value with the variable’s name with that allocated space.
* **static** is often used with **final** to allow access to the variable from other classes without an Object reference.
  + **final =** Assigned value is its only value
* **Accessor Methods** (Getter Methods)
  + Allow for queries to return data from a class that is stored in an instance variable
    - Layout:
      * **Scope** = Public
      * **Type** = Whatever type is required
      * **Name** = getInstanceVariable
      * **Parameters** = Whatever is required (usually none)
      * **Body** = return this.instanceVariable
* **Modifier Methods** (Setter Methods)
  + Allows for commands to modify instance variables
    - Layout:
      * **Scope** = Public
      * **Type** = Whatever type is required (usually void)
      * **Name** = setInstanceVariable
      * **Parameters** = Whatever is required
      * **Body** = this.instanceVariable = Argument
* **Encapsulation** = Fundamental OOP concepts
  + Method of wrapping data (variables) and code acting on the data (methods) together as a single unit
  + Variables of a class will be hidden from other classes and can only be accessed through the methods of their current class
    - Known as **Data Hiding**
  + To achieve Encapsulation in Java
    - Declare the variables **private** (to limit their scope to the current class)
    - Provide public accessor and modifier methods for these variables