Unit 2 Notes

9/18/23 - Complex Logic and Composition

Complex Conditionals

- a AND b - all must be true
- a OR b - at least one must be true
- a NOT b - negates the original condition

Logic Operators

- Logical and - `&&`
  - All must be true
- Logical or - `||`
  - Any statement can be true
  - Continues checking even if false

Precedence rules for arithmetic, logical, and relational operators

Operator	Description	Explanation
`()`	Items within parentheses are evaluated first	In `(a * (b + c) - d)` the + is first
`!`	`!` logical not is next	`! x || y` is evaluated as not x or is y
`< <= > >=`	Relational Operators	`x<2 || x>=10` is evaluated x<2 or x≥10
`== !=`	Equality and inequality	`x == 0 && x != 10` is evaluated as x is 0 and x is not 10
`&&`	Logical AND	`x == 5 || y == 10 && z != 10` is evaluated as x = 5 or y =

Operator	Description	Explanation
		10 and z ≠ 10
`||`	Logical OR	`||` has the lowest precidence of the listed arithmetic, logical, and relational operators

Composition

- A composition in Java between two objects associated with each other exists when there is a strong relationship between one class and another
- For example, one class has an attribute that is an object of another class

---

9/20/23 - Classes, Objects, and Composition

Class

- Describes a set of objects with the same behavior
    - String class describes the behavior of all strings
        - Specifies how a string stores its characters
        - Which methods can be used (behaviors)
        - How the methods are implemented
- Overloaded Constructors
    - Same name of method with different parameters
    - Each method has a different implementation
- We can have overload of any method, constructors or any other method defined in the class
- Has a public interface
    - Collection of methods through which the objects of the class can be manipulated
- Stores its data in instance variables (attributes)

    - Data required for executing the methods
    - Instance variables should always be private
    - Private instance variables can only be accessed by the method of its own class

Classes have variables / data

- Classes can have any number of variables and types in them
    - Variables in the class code block
        - May be static
            - Variable value is shared across all classes / the program
        - May be instance (not-static)
            - Variable value is only set for every instance / object uniquely
                - Length of the String only makes sense for unique strings!
    - Variables may have scope
        - Who has access to read them
        - `public` - everyone in every class can read and write to them
        - `private` - only methods in that class can read and write to them (suggested)
    - Can be changed or cant be changed
        - `final` - defines a constant, immutable, value

9/22/23 - Datatypes and Wrapper Class

Binary

- Binary - Two state system
    - 1 for on
    - 0 for off
- Bit

    - Each 0 and 1 is called a bit
    - 8 bits is called a byte
    - Contains 255 states (128+64+32+16+8+4+2+1)
- Every bit is an exponential of 2
    - 1 + 2 + 4 + 8, etc

Numeric Data Types

Declaration	Size	Supported Number Range
`byte myVar`	8 bits	-128 to 127
`short myVar`	16 bits	-32,768 to 32,767
`int myVar`	32 bits	-2,147,483,648 to 2,147,483,647
`long myVar`	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float myVar`	32 bits	-3.4×10^{38} to 3.4×10^{38}
`double myVar`	64 bits	-1.7×10^{308} to 1.7×10^{308}

Wrapper Class

- A *primitive* type variable directly sotre sthe data for that variable type, such as int, double, or char
- A *reference* type variable can refer to an instance of a class, also known as an object
- *Wrapper classes* that are built-in reference types that augment the primitive types

Reference Type	Associated primitive type
Character	`char`
Integer	`int`
Double	`double`
Boolean	`boolean`
Long	`long`

Wrapper Class Conventions

- Autoboxing - automatic conversion of primitive types to the corresponding wrapper classes
- Unboxing - automatic conversion of wrapper class objects to the corresponding primitive types

Character Wrapper Class

- char is a primitive
  - No methods by itself
- Character “wrapper” exists
  - Methods (mostly static) to help you learn about char
- Common and useful methods
  - Character.isDigit(char)
  - Character.isWhitespace(char)
  - Character.isLetter(char)
- These are often paired with String charAt, and loops

9/25/24 - ArrayList

Collections

- A collection represents a group of objects, known as its elements
- Some collections allow duplicate elements and others do not. Some are ordered and others unordered
- It is an interface
  - We will learn more later
  - For now, an interface specifies a set of behaviors (methods) that other classes needs to implement

List

- An ordered collection (also know as a sequence)
- The user of this interface has precise control over where in the list each element is inserted
- The user can access elements by their integer index (position in the list), and search for elements in the list

ArrayList

- Resizable-array implementation of the List interface
- Each ArrayList instance has a capacity
- Capacity is the size of the array used to store the elements
- When an instance of an ArrayList reaches its capacity, the instance grown automatically (resize), by doubling its initial capacity

```
ArrayList<Box>  boxList = new ArrayList<>();

boxList.add(new Box(10,3,4));
boxList.add(new Box(5,5,5));
boxList.add(new Box(10,3,13));

for (Box box : boxList) {
  System.out.println(box);
}

System.out.println(boxList.size());

boxList.remove(2);
```

Wrapper Classes

- ArrayList
  - Only stores objects
- Wrapper classes to the rescue
  - int has integer
  - double has Double

    - boolean has Boolean
    - char has Character
    - etc.
- Boxing and Unboxing
    - Allows automatic conversion between wrapper and primitive
    - Integer myInt = 10;
        - Same as Integer myInt = new Integer(10);

ArrayList: Some methods

- `.get(int)` - returns the item at the set index
- `.remove(int)` - removes the item at the set index (giving the item if needed)
- `.set(int, value)` - replaces an item at an already existing index
- `.add(value)` - adds an item to the end of the list
- `.add(int, value)` - inserts an item at a set location
- `.size()` - returns the total number of elements in the list
    - Yes this one is confused with `.length()` all the time

For Each

`for(Type element: list)`

- Get all the elements that are in a list

```
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
list.add(2);
list.add(-1);

for (Integer element: list)
  System.out.println(element)
```

9/27/23 - More Methods

Method Overloading

- You can have the same method name, different parameters
- Java will match the parameters on which method is called
- Best practice
  - Methods with less parameters call the most detailed version
  - This let's you have default values for methods
  - Makes it so you only have one place to update!

Reminder: Keep it simple

- Methods are the conquer: divide → conquer → glue
  - Which means, the smaller the problem to solve, the better
  - Keep what you do in a method simple
  - If you write 20 lines, you probably have written too much
  - If you cut and paste you need a method
- Turn problems into questions
  - Whaat is your quest?
  - What do you know?
  - What do you need?

Debugging

- println/print
  - one of the best tools to debug code
  - Not sure how something works, toss it in a print and see what happens
  - Unsure where the error is? put it in print lines to narrow down the issue
  - Suggestion use "TESTING" with all the statements, so you don't accidentally leave it in

- Debuggers
  - IDEs come with debuggers that help you trace code. Future thing to remember
- assert
  - New command
  - Good for testing completed methods (unit testing)
  - Operator that prints an error message and exists the program if the provided test expression evaluates to false
  - What you have been doing in labs is perfect for it

    `assert testExpres : detailedMessage;`

9/29/23 - String Manipulation

Strings

- A string is a collection of ordered characters
  - It has data
  - It has Functionality (methods)
  - It is also immutable (cant be directly modified)
    - Every Method that builds a String, returns a copy
    - Java does this for memory efficiency
- Example
  - `String mascot = "Cam";`

Common String Methods

- `charAt(int)` - gives us the character at location
- `indexOf(char)` - gives us the location of character (what you just wrote)
- `indexOf(String)` - overloaded option gives the location of the start of the string that matches

- `lastIndexOf(char)` - gives us the index starting at the end working down (also has a String version)
- `subString(int start, in end)` - returns the substring from start - including start to end, excluding end (inclusive / exclusive)
- `toLowerCase()` - returns the lowercase version of the string
- `toUpperCase()` - returns the uppercase version of the String

10/2/23 - Reading Files

I/O Streams

- A Stream is a sequence of data
- A program uses an input stream to read data from a source, one item at a time
- A program uses an output stream to write data to a destination, one item at a time

OutputStream

- `System.out` is a predefined OutputStream object reference that is associated with a system's standard output, usually a computer screen
- The `print()` and `println()` methods are overloaded in order to support the various standard data types

InputStream

- `System.in` is an input byte stream
- When using an InputStream, a programmer must append the clause `throws IOException` when using the method read()
    - The `throws` clause tells the JVM that the corresponding method may exit unexpectedly due to an exception, which is an event that disrupts a program's execution
- Instead of reading a byte stream, dealing with IOException, and after it converting the data to a String or other types, we have been using the Wrapper class named

Scanner

Scanner - Reading from a String

- When we want to read something from the terminal we use

```
Scanner scnr = new Scanner(System.in);
```

  - The parameter `System.in` indicates that we are reading from the terminal

- Sometimes we may want to read something from a String so instead of using System.in as a parameter we have a String as a parameter

Scanner - Different Uses

- We know how to read from terminal
- We know how to read from String
- What happens if instead of reading from the terminal or string, we want to read from a file
  - We need to use the `File` class
  - And pass an object of File as a parameter instead of System or a string when constructing Scanner

```
Scanner fileIn = new Scanner(new File(filename));
```

10/6/23 - More Loops

Incriminator and Decrementor

- `++` or `--` BEFORE the variable means
  - Add or subtract by 1 and then use the modified value
- `++` or `--` AFTER the variable means

    - Use the value in the variable, and then add or subtract by one modifying the value

Do While Loop

- A unique loop
    - For and while - both check and then run
- Do-while
    - Runs block of code and then checks
    - Guarantees at least one run

```
do {
  // some code
} while (true /* condition */);
```

Changing loop order: Break / Continue

- `break`
    - Allows us to beak out of the loop completely (early exit)
- `continue`
    - Allows us to move back to the top of the code block
    - with for loop the incriminator is still completed

10/9/23 - Inheritance

- Is a relationship between a more general class (called **superclass**) and a more specialized class (called **subclass**)
- The subclass inherits data and behavior from the superclass

Substitution Principle

- Substitution principle states that you can always use a subclass object when a superclass object is expected

Inheritance Example

```
public class Question {
  private String text;
  private String answer;

  public Question() {
    // calls the constructor with 2 params
    this("", "");
  }

  public Question(String text, String answer) {
    setText(text);
    setAnswer(answer);
  }

  public void setText(String text) {
    this.text = text;
  }

  public void setAnswer(String answer) {
    this.answer = answer;
  }

  public String getText() {
    return text;
  }

  public String getAnswer() {
    return answer;
  }

  public boolean checkAnwer(String answer) {
    return (this.answer.equals(answer));
  }
```

Inheritance - Makes Java DRY

- Inheritance
  - Heart of OOP
  - Essential to large programs
  - DRY
- A class can extend another class

    - By extending
        - Inherit methods and properties
    - override
        - Allows you to change methods for children

Object Class

- All classes in java extend Object
- Object is a type / class
    - Includes common methods
    - `toString()`
        - Returns String of the object
        - by default memory location (not useful) - should override
        - `System.out.println()` - calls toString()
        - String concatenation calls toString()
    - `equals(Object)`
        - Compares memory locations
        - should usually override

Revisiting Scope

- public
    - Everyone has access
- private
    - Only the class has access
    - This means child classes - can't access private
- protected
    - child class has access only
- <blank/omitted>

    - package and children have access

Inheritance is Polymorphic

- Substitution principle states that you can always use a subclass object when a superclass object is expected
- Children may appear to be their parents!
- Define a data structure of the parent type and you can store parent and children types
- Calls correct class
