Unit 3 Notes

10/16/23 - Operating Systems and File Output

Operating Systems

- You use them daily
  - Most common os in the world - Android
    - Written in Java - Kotlin
- The control
  - Resources
  - Hardware interaction
  - Devices
  - Running applications, memory, etc
  - Files

File Systems

- Program that helps manage files and other programs
- Directory Structure
  - Relative
    - Based on current location
  - Absolute
    - Based on Root, the top of the hierarchy
- Key “shortcuts”
  - `.` (yes dot) - current directory
  - `..` (directory above current)
- Examples

- Windows: drive letter with C:/, D:/, etc
- Linux/MacOS/Unix: just a “/”

File Object in java

- Has a number of useful methods when dealing with files and directories
- `File myFile = new File("filename");`
  - Creates or reads a file based on the path + filename given
  - Actually connects to the location which is a stream of bytes

FileOutputStream Object in Java

- Has a number of useful methods when dealing with writing binary data to a file
- `FIleOutputStream myFile = new FileOutputStream("output.txt");`
  - Creates a file in the same directory as the java executable. Relative

Print Writer

- `PrintWriter` is an object designed to write text to a File Stream
- What are streams?
  - `System.out` - Stream to the console
  - `System.in` - Stream from the console
  - `System.err` - stream to the error log (often console)
  - File is also a stream
  - FileOutputStream is also a stream
- PrintWriter uses the same interface as System.out but directs the stream

```
PrintWriter writer = new PrintWriter(new FileOutputStream("notes.txt");
writer.printn("#these are my notes");
```

10/18/23 - Java Exceptions

What are Exceptions

- Classes / Objects
  - They contain information about the error that is happening
- What about try / catch and throws?
  - Those are commands that use those objects
- `try / catch`
  - `try {}` - Says “try this block of code”
  - `catch (Exception x) {}` - Run this block of code if there is an error
  - `finally {}` - always run this block of code error or not (often can be omitted, won’t be used much in this class)

Java Exception Hierarchy

- Error class is used to indicate a more serious problem in the architecture and should not be handled in the application code.
- Exception class is used for exception conditions that the application may need to handle.
- Exceptions are further subdivided into checked (compile-time) and unchecked (run-time) exceptions
- Exceptions that can occur at compile time are called checked exceptions since they need to be explicitly checked and handled in codde
  - All classes with the exception of Error and RuntimeException are checked
- Unchecked exceptions can be thrown “at any time” (i.e. run-time). Therefore, methods don’t have to explicitly catch or throw unchecked exceptions
  - RuntimeException

Controlling Exceptions

- We can’t control every possible error situation

- For example
  - What happens if the file is not there?
  - What if you don’t have permission to read it?
  - Not just files
    - What about network connections?
    - What if printers aren’t there?
- Exception handling
  - try catch

Try - Catch

- `try`
  - Try a block of code
  - if it runs properly, great!
- `catch`
  - An exception happened!
  - run the catch block of code
- `throws`
  - Allows you to throw the exception
  - requires someone else to handle it
- `Exception`
  - An object/class we use for errors
  - You can write your own
    - or use build-in cases
  - Checked (compile time) or Unchecked (run time)
    - Checked requires try / catch

Advanced: Creating your own exception

- You can create and throw your own exceptions (often called “raise” in other languages)
- In Java, you have to extend the Exception class to do that
  - Ensures certain methods are implemented for try / catch / throw / throws
- Won’t use it much in this class but still worth knowing
- Especially if you are developing an SDK or API

10/20/23 - More Classes

Static x Instance Variables

- Static
  - Belongs to the class
  - How do you access a public static variable outside of the class?
    - ClassName.staticVaraibleName
  - Example
    - `Cake.IS_GOOD`
- Instance
  - Belongs to the object
  - How do you access a private instance variable?
    - You will need to have a get method for each variable that you want to have access from other class
    - ObjectName.getNameVariable()
  - Example
    - `Cake.getName()`

Static Methods

- Instance methods

    - Methods that need class information
- Static method
    - Methods that are “self-contained”
    - Matches the concept of a class but is not unique to the object
    - Static may not call instance methods without building an object
        - But instance can call static

Overloaded Constructors

- Just like methods
    - constructors can be overloaded
- Standard practice
    - call the most specific constructor with default values
    - `this()` is used to call the constructor
    - must be the first line of the constructor
    - keep it dry
- When you write a constructor with parameters the default one is not supported anymore
- Really ask yourself
    - What do you need
    - where do you get it

Packages

- Is a grouping of related types, classes, interfaces, and sub-packages
- Use “import” to add those packages to your program
- `java.lang` is automatically imported into all Java programs
- `import java.io.File;` VS `import java.io.*;`

Unit Testing

- A program whose job is to thoroughly test another program (or portion) via a series of input/output checks known as test case]

10/23/23 - Polymorphism & More Branching

Composition

- Has-a relationship

```
public class ChildInfo {
  public String firstName;
  public String birthDate;
  public String schoolName;

  ...
}

public class MotherInfo {
  public string firstName;
  public string birthDate;
  public string spouseName;
  public ArrayList<ChildInfo> childrenData;
  ...
}
```

Inheritance

- Is-a relationship

```
public class PersonInfo {
  public String firstName;
  public String birthDate;
  ...
}

public class ChidInfo extends PersonInfo {
  public String schoolName;
  ...
}

public class MotherInfo extends PersonInfo {
  public String spouseName;
```

```
  public ArrayList<ChildInfo> childrenData;
  ...
}
```

Polymorphism

- Refers to determining which program behavior is executed depending on data types
- Polymorphism of methods - methods overloading
  - compile-time polymorphism
  - compiler determines which of several identically-named methods to call based on the method’s arguments
- Polymorphism of variables - involves derived classes (inheritance)
  - runtime poly
  - compiler cannot make the determination but instead the determination is made while the program is running

Polymorphism of variable

- Substitution principle - you can always use a subclass object when a superclass is expected
- Super class variable can store super class types and sub class types as well
- Sub class variable can only store sub class types

ArrayList of Objects

- Store a collection of objects of various class types

instanceof

- Used to determine the type of an object

```
public static void printArrayListV2(ArrayList<Object> objList) {
  int i;
  for (i=0; i<objList.size();i++) {
    Object obj = objList.get(i);
    if (obj instanceof String) {
      // do something
```

```
    }
  }
}
```

Conditional Statements vs. Ternary Statements

- Conditional - if else else if

```
if (true) {
  // do something
} else {
  // do something else
}
```

- Ternary ? :

```
String time = 10 > 5 ? "Hello" : "Goodbye";
// results in time = "Hello"
```

Switch Statements

- switches
  - a condition that checks each “case” for using ==
  - concise way to compare against a group of options
- `case`
  - the cases to ==
- `break`
  - keeps executing code until break is called

```
switch (primitive or String) {
  case <value>:
    // do something
  case <value>:
    break
  default: // else
}
```

Enumerations

- Declares a name for a new type and possible values for that type
- Methods can use them and return them!

Switch + Enum

- Switch + enumerations are strong combinations
- Enumeration is part of the case

10/25/23 - More Branching

Conditional Statements / Ternary Statements

```
if (true) {
  // do something
} else {
  // do something else
}
```

- A way to write a simple if / else on one line

  `condition ? value if true : value if false`

  ```
  String time = 10 > 5 ? "Hello" : "Goodbye"
  ```

Switch Statements

- Switches
  - A condition that checks each “case” for using ==
  - A concise way to compare against group of options
- Case
  - the cases to ==
- Break

- Keeps executing code - until break is called

10/27/23 - Arrays

Recalling to the past

- For every value you want to store, you need a value
- What if you want to store 100 values? 10,000 values?
  - Use ArrayLists for storing objects
  - But how about primitive types
    - Introducing Arrays
    - Reserves memory for storing values, in order from the 0 index
- Sound familiar - Recall string
  - the String object contains chars in order
  - It is a character array!

Arrays are mutable

- Elements in the array can be changed / reset!

Arrays can be any type!

- `int[] values = new int[100];`
- `String[] names = new String[10];`
- Format is
  - `TYPE[] name = new Type[size];`

Array Length

- Array size allocated to size 10
  - 0…9 indicies valid

- rhps[20]
  - throws `IndexOutOfBoundsException`
- How to check for that?
  - `array.length`
  - Notice no parenthesis, command, not a method

Arrays vs. ArrayLists

- ArrayLists are lists that use Array as the underlying structure
- Arrays are just how you declare a group of objects in order
- ArrayList is an individual object someone wrote

When to use arrays over array lists?

- When your size is fixed, arrays are much faster to use!
- When you need to keep order on sparsely populated datasets (that are often fixed sizes)
  - [value, null, null, null]
- They are used about equally, just depends on what you are doing

---

10/30/23 - Intro to Recursion

Recursion

- Simple recursion is a loop
  - a Method that calls itself
- How to write it?
  - Write a base case (condition)
  - Write a method that calls itself
- Starting with factorial example

```
public static long factorialLoop(int n) {
  long fact = 1;
  for (int i = n; i > 1; i--) {
    fact *= i;
  }
  return fact;
}
```

- Building our first recursion method
  - Factorial
  - 0! = 1
  - 1! = 1
  - `n != n*(n-1)*(n-2)*...*2*1`

Recursion - String reverse

- Let's start with a String reverse method that uses a loop solution to return a reversed string

```
public static String reverseLoop(String str) {
  String reveresed = "";
  for (int i=str.length(); i>-1; i--) {
    reversed += str.charAt(i);
  }
  return reversed;
}
```

- How to write a recursive version of it
  - Write a base case `if (index<0) return "";`
  - Write method that calls itself
    - `return str.charAt(index) + reverseString(str, index-1);`

Real example + sneak peak into inheritance

- Assume you have the following data structure
  - An array of values, but some values can be other arrays

- How do you represent all the different values?
- Inheritance (and polymorphism)
  - All objects “inherit” from the class object
    - Gains the properties of Object
    - Which means you can store all objects as objects
    - But you need to “cast” back to do something useful
- Now - The real example
  - How can I sum all the values across the structure to the right?
    - Would a loop work?
      - No!
  - Solution… Recursion!

```
public cass Recursion {

  public static int sum(Object[] values) {
    return sum(values, 0); // overloaded
  }
  public static int sum(Object[] values, int current) {
    if (current >= values.length) reutrn 0; // past the end of array
    if (values[current] instanceof Object[]) { // another array
      return sum((Object[])values[current], 0) + sum(values, current+1);
    }
    return (Integer)values[current] + sum(values, current + 1); // number plus something
  }
  public static void main(String[] args) {
    Object[] values = new Object[]{1,2,3,
      new Object[]{4,5, new Object[]{1,1}},
      10, new Integer[]{2,2}, 1, 10};
    System.out.println(sum(values));
  }
```

11/1/23 - Abstract Classes and Interfaces

Inheritance

- Creates an is-a relationship between classes
- Allows fully implemented “more generalized” classes as the super classes
  - Specialized subclasses as the subclasses inherits instance variables and methods from the super class
- Use the key word `extends`
  - Can only extend / inherit from one immediate parent (but can have “chain” of parents)

```
Circle crcl = new Circle(10, "234,255,123");

System.out.println("the color is " + crcl.getColor());
```

Polymorphism

- Allows the subclass to be declared as the super
  - Actually a subclass can “substitute” in for the super
- Extremely useful for things like Arrays and ArrayLists

```
Shape[] shapes = new Shape[3] // fixed size
shapes[0] = crcl;
shapes[1] = new Rectangle(23, 5, "123,125,255");

System.out.println(Arrays.toString(shapes));
```

Abstract class

- What if a method you write needs specific information?
  - Unique to subtypes / children
  - BUT - the rest of the method is general
- Enter Abstract Classes
  - Classes that are not *complete* by themselves
  - Contains **partial** implementations of a class

        - With other methods that are required to be completed by children
- Class is abstract, some methods are abstract
- Can’t be instantiated
    - But can have constructors the children can inherit

Interfaces

- Inheritance Limitation: Can only inherit **one** class directly
    - meaning, there can be a chain of classes
- What if we wanted to inherit from more than one class?
- Enter interfaces
    - contracts that define what methods will be implemented
    - Contains no implementation - just definitions
    - uses implements in the class to say class is following contract
- Common Interface
    - comparable
        - Implementing it - allows objects to be sorted in ArrayList!
        - Compare to is the method