Zybooks Summary

Table of Contents

Chapter 1 - Intro to Java

Ex. a simple program with user input and print output

```
import java.util.Scanner;

public class Salary {
   public static void main(String [] args) {
      int wage;

      Scanner scnr = new Scanner(System.in);
      wage = scnr.nextInt();

      System.out.print("Salary is ");
      System.out.println(wage * 40 * 52);
   }
}
```

- A ***program*** starts in main() { … }
- Could combine as `int wage = 20;`
- `println` adds newline after output
- `System.in` refers to keyboard
- Import packageName.className like java.util.Scanner

Chapter 2 - Objects

- ***object***: internal data items plus operations that can be performed on that data.
- private: internal data are private to the object
- public: operations that can be called on an object are public methods, can be called from outside the object

Ex. Creating new object and method call

```
PeopleCounter attendeeCounter = new PeopleCounter();
Public void setCount(int count){...}

attendeeCounter.setCount(count);
```

- object attendeeCounter in a class named PeopleCounter
- setCount(parameter) method on an object named attendeeCounter
    - objectName.methodCall(argument)
    - void means method won't return value (it's a setter)

- Arithmetic order of operations:
  - ( )
  - negative -
  - * / %
  - + -
- Compound operators -=, *=, /=, and %=, +=

- Comments: // (single line) or /* … */ (block)

Ex. Attributes, Constructor, Methods

```
//attribute list
      private double fuelCapacity;
      private int numOfDoors;
//Constructor
   public Car(double capacity){
      fuelCapacity = capacity;
      numOfDoors = 4;
   }
//method list
      public void setFuelCapacity(double capacity) {
         fuelCapacity = capacity;
      }
      public double getFuelCapacity(){
         return (fuelCapacity);
      }

Car myCar = new Car(16)
```

- Attributes usually private
- A method may return one, and only one, value using a ***return statement***
  - `getFuelCapacity()` defined to return a `double` so return statement must evaluate to a double
- The constructor has the same name as the class. The constructor method has no return type, not even void
  - need parentheses after new object to call constructor (even if default)
  - when you have 2 constructors with same parameter types, write in diff order

Chapter 3 - Methods & Classes

- Methods must be defined within a class
- Static fields/methods vs instance variables/methods

- main() method can be defined within a programmer-defined class and create objects of that class type
    - main() is a static method
        - independent of class objects
        - can access other static methods and static fields of the class, but cannot directly access non-static methods or fields
            - must create objects within main() to call non-static methods on those objects.
    - Ex: BasicCar's main() creates two objects of type BasicCar and performs operations on those objects.

Ex. implicit parameter 'this'

```
private double sideLength;
public void setSideLength(double sideLength) {
   this.sideLength = sideLength;
}
```

- `this.sideLength` refers to field member, `sideLength` is parameter
- Use if parameter and field member have same identifier

Chapter 4 - Data Types

- Ex. final variable: `final double SPEED_OF_SOUND  = 761.207;`
- Math class methods include
    - `math.sqrt(x)`
    - `math.pow(x, y)  //x to the y power`
    - `math.abs(x)`
- Integer division has no fractions (be careful that the answer is what you want)
- Type conversions usually automatic if no loss of precision (ex. int to double)
    - Explicit type cast ex. `myInt = (int)myDouble`
        - fractional part is truncated, i.e. 9.5 becomes 9
- Character literal ex. `char myChar = 'm';`
- Character from input: `myChar = scnr.next().charAt(0);`

    - Gets first character from next non-whitespace (string) input
- Characters are stored as ASCII (technically Unicode) numbers
    - Capital letters < lowercase
- Escape sequences include (ex. `myChar = '\''`):
    - `\n` (newline)
    - `\"` (quotes)
    - `\t` (tab)
- Multiple character variable outputs
    - Ex. `System.out.print("" + 'a' + 'b');`
        - Initial `""` tells compiler to output string of characters (otherwise compiler would add numerical values of 'a' and 'b', i.e. 195 instead of ab

- Getting String input without whitespace
    - `userString = scnr.next();`
        - First non-whitespace up to next whitespace (doesn't touch the newline)
        - i.e. "Hi there!" would return "Hi"
- String input with whitespace
    - `scnr.nextLine()` gets all remaining text on the current input line, up to the next newline character
    - Even if all that's left in line is newline, it will take that
- Other types of input (byte, short, int, long)
    - `myInt = scnr.nextInt();`

Ex. Generating random number

```
import java.util.Random;

public class ThreeRandomValues {
   public static void main(String[] args) {
      Random randGen = new Random();  // New random number generator

      System.out.println(randGen.nextInt());
      System.out.println(randGen.nextInt());
      System.out.println(randGen.nextInt());
   }
}
```

- ***Random*** class provides methods that return a random integer in range
- ex. random integer in the range 18 to 30: `randGen.nextInt(30 - 18 + 1) + 18`
- A programmer can specify the seed when the Random object is created or using the setSeed() method. With a specific seed, each program run will yield the same sequence of pseudo-random numbers.
    - ex. `Random randGen = new Random(5);,`
    - ex. `randGen.setSeed(5);`

Chapter 5 - Branches

Ex. If-else branch

```
if (numYears == 1) {
       System.out.println("Your first year -- great!");
}
else if (numYears == 10) {
       System.out.println("A whole decade -- impressive.");
}
else {
       System.out.println("Nothing special.");
}
```

- else is usually optional unless if statement doesn't provide output when output is required

- Detecting equality (integer, character, float)
  - ==, !=
  - Evaluates to a boolean
  - Comparing characters compares their Unicode numerical encoding
  - Avoid comparing floats with these equality operators
    - imprecise representation of floating-point numbers
  - These operators should not be used with strings
- Relational operators: <, >, <=, >=
- logical operators: && , || , !
- Precedence for operators:
  - ( )
  - !
  - Arithmetic operators (* / % + -)
  - < <= > >=
  - == !=
  - &&
  - ||

Ex. Conditional expression

```
if (condition) {myVar = expr1;}

else {myVar = expr2;}
```

Becomes

```
myVar = (condition) ? expr1 : expr2
```

- Form is `condition ? exprWhenTrue : exprWhenFalse`
- Good practice: restrict use to assignments

Ex. Switch statement

```
switch (expression) {
  case 0:
     ...
     break;

  case 1:
     ...
     Break;

  default:
     ...
     break;
}
```

- Compares variable (integer, char, or string) to constant values
- executes the first ***case*** whose constant expression matches the value of the switch expression
- If no case matches, then the ***default case*** statements are executed
- Omitting `break` causes falling through to next case

- String comparisons
  - `str1.equals(str2)` returns boolean just like ==
  - `str1.compareTo(str2)` returns numbers
    - negative (str1 < str2), 0 (str1 = str2), positive (str1 > str2)
      Ex. `if (myName.compareTo(yourName) > 0 { … }`
      - true if compareTo() returns a positive number (if myName > yourName)
  - Compare string while ignoring case
    - `str1.`***`equalsIgnoreCase`***`(str2)`
      `str1.`***`compareToIgnoreCase`***`(str2)`
- String operations (note Strings are immutable so may need to generate new String)
  - `s1.charAt(x)` determines the character at index x of a string
  - `s1.length()` returns string length
  - `s1.concat(s2)` or `+` appends s2 to s1
  - `s1. indexOf(item:char/String variable/literal)` gets index of first item occurrence in a string, else -1.
    - `s1.indexOf(item, indx)` looks for `item` starting at index `indx`.

        - `s1.lastIndexOf(item)` finds the last occurrence of the item in a string, else -1.
    - `substring(startIndex)` returns substring starting at startIndex.
        - `substring(startIndex, endIndex)` returns substring starting at startIndex and ending at endIndex - 1
        - The length of the substring is endIndex - startIndex
    - `replace(findStr, replaceStr)` returns a new String in which all occurrences of findStr have been replaced with replaceStr
        - `replace(findChar, replaceChar)` returns a new String in which all occurrences of findChar have been replaced with replaceChar
- Character class operations
    - `Character.toUpperCase(let0)`
    - `Character.toLowerCase(let0)`
    - `Character.isLowerCase(let0)`
    - `Character.isUpperCase(let0)`
    - `Character.isLetter(let0)`
    - `Character.isDigit(let0)`
    - `Character.isLetterOrDigit(let0)`
    - `Character.isWhitespace(let0)`

Chapter 6 - Loops

Ex. while loop

```
{
   int i = 0;
   while (i < 5) {
      //loop body;
      ++i;
   }
}
```

- Each execution of the loop body is called an ***iteration***.
- Once entering the loop body, execution continues to the body's end, even if the loop expression becomes false midway through.

Ex. do-while loop

```
do {
```

```
   // Loop body
} while (loopExpression);
```

- A ***do-while loop*** first executes the loop body's statements, then checks the loop condition.
- useful when the loop should iterate at least once.

Ex. for loop

```
for (int i = 0; i <= 10 ; ++i) {
  // Loop body
}
```

- Useful when number of iterations is known
- Good practice is to declare loop's index variable in the initialization statement
- Initialization statement: variable initialization, loop condition, variable update.
- ++ is ***increment operator*** (***decrement operator*** is --).
    - ++i (increments, then assigns) vs. i++ (assigns, then increments)
        - Outputs may differ, but not always
        - Ex: If i is 5, outputting ++i outputs 6, while outputting i++ outputs 5 (and then i becomes 6).

Ex. break statement causes immediate exit of the loop

```
if (mealCost == userMoney) {
                break;
            }
```

Ex. continue statement causes immediate jump to the loop condition check

```
if (((numTacos + numEmpanadas) % numDiners) != 0) {
                continue;
            }
```

- ex. If the number of tacos and empanadas is not divisible by the number of diners, the continue statement proceeds to the next iteration, thus causing incrementing of numEmpanadas and checking of the loop condition.

Chapter 7 - Arrays

Ex. array

```
public class ArrayExample {
```

```
  public static void main (String [] args) {
     int[] itemCounts = new int[3];

     itemCounts[0] = 122;
     itemCounts[1] = 119;
     itemCounts[2] = 117;

     System.out.print(itemCounts[1]);
  }
}
```

- An ***array*** is an ordered list of items of a given data type.
- The array declaration uses [ ] after the data type to indicate that the variable is an array reference.
- Each item in an array is called an ***element***.
- `int[] itemCounts = new int[3];` declares an array reference variable itemCounts, allocates an array of 3 integers, and assigns itemCounts with allocated array.
- Indices begin with 0 and must be an integer
- An array's length can be accessed by appending .length after the array's name
  - Ex: `userVals.length` gives number of elements in userVals
- A programmer may initialize an array's elements in braces {} separated by commas.
  - Ex: `int[] myArray = {5, 7, 11};` creates an array of three integer elements with values 5, 7, and 11.
    - array's size is automatically set to the number of elements within the braces so it does not require the use of the new keyword

- To swap elements of an array, create a temporary variable to store one of the elements
- When copying, subtracting, etc. between 2 arrays, be careful not to call out-of-range values when array sizes are different
- Arrays' length cannot be changed. It must be copied to an array of a different size instead.

- 2-dimensional arrays
  - `int[][] myArray = new int[R][C]` represents a table of int variables with R rows and C columns, so R*C elements total.
  - Example accesses are `myArray[0][0] = 33;` or `num = myArray[1][2].`
- You can think of them as a table with rows and columns but it's really an array of arrays.
  - Ex: `int[][] myArray = new int[2][3]` allocates a 2-element array, where each array element is itself a 3 element array.
- A programmer can initialize a two-dimensional array's elements during declaration
  - To traverse all elements, use loop within loop

System.out.println(""); }

```
int[][] numVals = {
      {22, 44, 66}, // Row 0
      {97, 98, 99}  // Row 1
};
```

- Arrays of three or more dimensions can also be declared, as in: `int[][][] myArray = new int[2][3][5]`, which declares a total of 2*3*5 or 30 elements

Ex. Enhanced for loop

```
String[] teamRoster = new String[3];
      teamRoster[0] = "Mike";
      teamRoster[1] = "Scottie";
      teamRoster[2] = "Toni";
System.out.println("Current roster:");

for (String playerName : teamRoster) {
   System.out.println(playerName);
}
```

Does equivalent of

```
for (int i = 0; i < teamRoster.length; ++i) {
   String playerName = teamRoster[i];
   System.out.println(playerName);
}
```

- Compared to a regular for loop, an enhanced for loop decreases the amount of code needed to iterate through arrays, thus enhancing code readability and clearly demonstrating the loop's purpose.
- Also prevents incorrectly accessing outside of array range.

Chapter 8 - Wrapper Classes & ArrayLists

- Getters (mutator) and setters (accessor)
- Private helper methods
    - can be called from any other method of the class, whether public or private
    - users cannot call private member methods

Ex. Private helper method

```
private void setHumanYears() {

    //insert if-else statements

   humanYears = years * factor;

}

public void setWeightAndAge(int weightToSet, int yearsToSet) {

   weight = weightToSet;

   years = yearsToSet;

   setHumanYears();

}
```

- Default constructor: constructor with no arguments
    - If a class does not have a programmer-defined constructor, then the Java compiler *implicitly* defines a default constructor with no arguments
    - If a programmer defines any constructor, the compiler does not implicitly define a default constructor, so one should be explicitly defined
- Overloading a constructor: differ in parameter types. When an object is created, the constructor with matching parameters will be called.

- ***Reference***: variable type that refers to an object.
    - may be thought of as storing the memory address of an object.
    - Variables of a class data type and array types are reference variables.
    - Ex. `String firstName;` declares a reference to an object of type String.
        - reference variables do not store data for class types. Instead, the programmer must assign each reference to an object, which can be created using the new operator.

        - Object and reference are separate things; you can have 2 references for one object (calling method on one reference updates the object for both)

Ex. 2 references can refer to same object

```
RunnerInfo lastRun;
RunnerInfo currRun = new RunnerInfo();

// Assign reference to lastRun
lastRun = currRun;
```

- Variables can be 2 types
    - ***primitive type*** (ex. int, double, or char) which directly stores the data for that variable type,
        - Ex: `int numStudents = 20;` declares an int that directly stores the data 20.
    - A ***reference type*** variable can refer to an instance of a class, also known as an object.
- ***wrapper classes***: built-in reference types that allow the program to create objects that store a single primitive type value. The wrapper classes also provide methods for converting between primitive types (e.g., int to double), between number systems (e.g., decimal to binary), and between a primitive type and a String representation.

Wrapper Class Reference type	Associated primitive type
Character	char
Integer	int
Double	double
Boolean	boolean

- Many of Java's built-in classes, such as Java's Collection library, only work with objects.
    - Ex. a programmer can create an ArrayList containing Integer elements `ArrayList<Integer> frameScores;` but not an ArrayList of int elements.
    - A programmer may use a wrapper class variable in expressions in the same manner as the primitive type int. An expression may even combine Integers, ints, and integer literals.
- A wrapper class object (as well as a String object) is ***immutable***
- Comparisons

    - Do not use equality operators == and != when comparing wrapper class variables (object variables) for same reason as Strings. Compares references to objects, not the value of the objects.
    - Equality operators are valid when comparing a wrapper class object with a primitive variable or a literal constant
    - The relational operators <, <=, >, and >= may be used to compare wrapper class objects. However, note that relational operators are not typically valid for other reference types.
- Reference variables of wrapper classes can also be compared using the ***equals()*** and ***compareTo()*** methods (examples below are Integer class, but also apply to the other wrapper classes).
    - The use of comparison methods is slightly cumbersome in comparison to relational operators, but you don't have to memorize exactly which comparison operators work as expected.

equals(otherInt)	true if both Integers contain the same value. otherInteger may be an Integer object, int variable, or integer literal.
compareTo(otherInt)	Returns: 0 if the two Integer values are equal, negative number if Integer value < otherInteger's value positive number if Integer value > otherInteger's value. otherInteger may be an Integer object, int variable, or integer literal. `num1.compareTo(num2) // Returns value greater than 0, because 10 > 8`

- Wrapper class conversions
    - ***Autoboxing*** is the automatic conversion of primitive types to the corresponding wrapper classes.
    - ***Unboxing*** is the automatic conversion of wrapper class objects to the corresponding primitive types.

- The Integer, Double, and Long wrapper classes provide methods for converting objects to primitive types.
    - `.intValue()`
    - `.doubleValue()`
    - `.longValue()`

- The Character and Boolean classes support the ***charValue()*** and ***booleanValue()*** methods, respectively, which perform similar functions
- Wrapper classes feature methods that are useful for converting to and from Strings. Several of these methods are static methods, meaning they can be called by a program without creating an object. To call a static method, the name of the class and a '.' must precede the static method name, as in `Integer.toString(16);`.
- Methods for wrapper classes
    - *toString*() returns a String containing the decimal representation of the value contained by the wrapper class object
        - Ex. if `Integer num1 = 10;` then `num1.toString() // Returns "10"`
        - Works with Integer/Double object, int variable/literal
    - *Integer.parseInt*(someString) parses someString and returns an int representing the value encoded by someString.
        - Static method
        - Also available for the other wrapper classes (e.g. `Double.parseDouble(someString)`), returning the corresponding primitive type.
    - *Integer.valueOf*(someString) parses someString and returns a new Integer object with the value encoded by someString.
        - static method
        - also available for the other wrapper classes (e.g. `Double.valueOf(someString)`), returning a new object of the corresponding type.

- Reference variables can be arguments to method parameters
    - argument's reference is copied to the parameter so that both refer to the same object

- Abstract data types ex. String, ArrayList

Ex. ArrayList

```
ArrayList<Integer> valsList = new ArrayList<Integer>();


valsList.add(31);

valsList.add(41);

valsList.add(59);


System.out.println(valsList.get(1));
```

```
valsList.set(1, 119);


System.out.println(valsList.get(1));
```

- ArrayList: ordered list of reference type items that comes with Java.
    - Each item in an ArrayList is known as an ***element***.
    - The statement `import java.util.ArrayList;` enables use of an ArrayList.
- Ex. `ArrayList<Integer> valsList = new ArrayList<Integer>();` creates reference variable valsList that refers to a new ArrayList object consisting of Integer objects.
- ArrayList list size can grow to contain the desired elements.
- ArrayList does not support primitive types like int, but rather reference types like Integer.
- Operations
    - `ArrayList.add(element or index,element)` adds elements to end of list or at `index` (and everything greater gets shifted right)
    - `ArrayList.get(index)` returns element at `index`
    - `ArrayList.set(index, element)` replaces element at `index` with `element`
    - `ArrayList.size()` returns number of list elements
    - `ArrayList.isEmpty()` Returns true if the ArrayList is empty
    - `ArrayList.clear()` Removes all elements from the ArrayList.
    - `ArrayList.remove(existingElement or index)` Removes the first occurrence of an element which refers to the same object as existingElement or removes element at specified index. Elements from higher positions are shifted back to fill gap. ArrayList size is decreased by one.
- An ArrayList is a **Collection** supported by Java for keeping groups of items.
    - Other collections include LinkedList, Set, Queue, Map, etc.
    - Select the collection whose features best suit the desired task. For example, an ArrayList can efficiently access elements at any valid index but inserts are expensive, whereas a LinkedList supports efficient inserts but access requires iterating through elements. So a program that will do many accesses and few inserts might use an ArrayList.

Ex. enhanced for loop/for each loop for ArrayList

```
ArrayList<String> teamRoster = new ArrayList<String>();
```

```
// Adding player names
teamRoster.add("Mike");
teamRoster.add("Scottie");
teamRoster.add("Toni");

System.out.println("Current roster:");

for (String playerName : teamRoster) {
   System.out.println(playerName);
}
```

- The ***enhanced for loop*** iterates through each element in array/Collection.
    - also known as a ***for each loop***.
    - declares a new variable that will be assigned with each successive element of a container object, such as an array or ArrayList.
    - only allows elements to be accessed forward from the first element to the last element.

Chapter 10 - Recursion

- ***recursive algorithm***: algorithm that breaks the problem into smaller subproblems and applies the same algorithm to solve the smaller subproblems.
    - ***recursive method*** calls itself
        - **Write the base case** -- Every recursive method must have a case that returns a value without performing a recursive call. That case is called the ***base case***. A programmer may write that part of the method first, and then test. There may be multiple base cases.
        - **Write the recursive case** -- The programmer then adds the recursive case to the method.
        - Ex. The following illustrates a simple method that computes the factorial of N (i.e. N!). The base case is N = 1 or 1! which evaluates to 1. The base case is written as `if (N <= 1) { fact = 1; }`. The recursive case is used for N > 1, and written as `else { fact = N * NFact( N - 1 ); }`.
- Make sure to cover all possible base cases and to always reach a base case. Such errors may lead to infinite recursion
- Typically, programmers will use two methods for recursion.

    - An "outer" method is intended to be called from other parts of the program, like the method `int calcFactorial(int inVal)`. The outer method may check for a valid input value, e.g., ensuring inVal is not negative, and then calling the inner method.
    - An "inner" method is intended only to be called from that outer method, for example a method `int calcFactorialHelper(int inVal)`. Commonly, the inner method has parameters that are mainly of use as part of the recursion, and need not be part of the outer method, thus keeping the outer method more intuitive.
- Passing an int variable as an argument to a method results in the parameter having a separate copy of the int variable's value. If this value is changed inside the method, the value of the argument will not be changed because primitive data types are passed by value.
    - Arrays behave differently in methods because a copy of the array reference is stored in the method stack frame, and the array elements are stored on the heap. The array references in the stack frames of calling and called methods refer to the same array and have access to the array elements. Both methods can use and modify array elements.

Chapter 11 - I/O, Exceptions, Interfaces

- Java provides a variety of built-in classes, such as Scanner, ArrayList, File, and many others, that programmers can use to write programs. Given the large number of built-in classes, Java organizes related classes into groupings called packages.
- A ***package*** is a grouping of related types, classes, interfaces, and subpackage. The types, classes, and interfaces in a package are called ***package members***. The following table lists several built-in Java packages and sample package members.
- Table 10.6.1: Common Java packages.

Package	Sample package members	Description
java.lang	String, Integer, Double, Math	Contains fundamental Java classes. Automatically imported by Java.
java.util	Collection, ArrayList, LinkedList, Scanner	Contains the Java collections framework classes and miscellaneous utility classes.
java.io	File, InputStream, OutputStream	Contains classes for system input and output.
javax.swing	JFrame, JTextField, JButton	Contains classes for building graphical user interfaces.

A programmer can use a package member using one of the following methods.

- *Using a package member's fully qualified name:* A class' ***fully qualified name*** is the concatenation of the package name with the class name using a period. Ex: `java.util.Scanner` is the fully qualified name for the `Scanner` class in the `java.util` package.
- *Using an import statement to import the package member:* An ***import statement*** imports a package member into a file to enable use of the package member directly, without having to use the package member's fully qualified name. Ex: `import java.util.Scanner;` imports the Scanner class into a file and allows a programmer to use `Scanner` instead of `java.util.Scanner`.
- *Using an import statement to import every member in a package:* A programmer import all members of a package by using the ***wildcard*** character * instead of a package member name. Ex: `import java.util.*;` imports all classes in the java.util package and allows a programmer use package members such as Scanner and ArrayList directly.

Ex. Output formatting

```
String account = "Prime";

double total = 210.35;

int years = 5;



System.out.printf("The %s account saved you $%f over %d years\n",

    account, total, years);
```

- Examples of format specifiers for the printf() and format() methods
  - %c - char - print character
  - %d - int, long, short - prints decimal
  - %f - float, double - prints float
  - %e - float, double - Prints a floating-point value in scientific notation
  - %s - String - Prints the characters in a String
- Formatting floating-point output is commonly done using sub-specifiers between the % and format specifier character.
  - `%(flags)(width)(.precision)specifier`
  - Width: Specifies the minimum number of characters to print. If the formatted value has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).
  - .precision: Specifies the number of digits to print following the decimal point. If the precision is not specified, a default precision of 6 is used.
  - Flags:

        - -: Left aligns the output given the specified width, padding the output with spaces.
        - +: Prints a preceding + sign for positive values. Negative numbers are always printed with the - sign.
        - 0: Pads the output with 0's when the formatted value has fewer characters than the width.
        - space: Prints a preceding space for positive value.
    - Method calls to printf() apply to PrintStream objects like System.out. These apply to ints and Strings too (excluding inapplicable methods like decimals for ints).

Ex. Input: Reading from String with Scanner

```
import java.util.Scanner;

public class StringInputStream {
   public static void main(String[] args) {
      Scanner inSS = null;              // Input string stream
      String userInfo;                  // Input string
      String firstName;                 // First name
      String lastName;                  // Last name
      int userAge;                      // Age

      userInfo = "Amy Smith 19";

      // Init scanner object with string
      inSS = new Scanner(userInfo);

      // Parse name and age values from string
      firstName = inSS.next();
      lastName = inSS.next();
      userAge = inSS.nextInt();

      // Output parsed values
      System.out.println("First name: " + firstName);
      System.out.println("Last name: " + lastName);
      System.out.println("Age: " + userAge);
   }
}
```

- The statement `Scanner inSS = new Scanner(userInfo);` creates a Scanner object in which the associated input stream is initialized with a copy of userInfo. Then, the program can extract data from the scanner inSS using the family of next() methods (e.g., next(), nextInt(), nextDouble(), etc.).

Ex. Processing User Input Line-by-Line (reads line as String, then extracts from String)

```
lineString = scnr.nextLine();
```

```
      // Create new input string stream
      inSS = new Scanner(lineString);

      // Now process the line
      firstName = inSS.next();

      // Output parsed values
      if (firstName.equals("Exit")) {
         System.out.println("   Exiting.");

         inputDone = true;
      }
      else {
         lastName = inSS.next();
         userAge = inSS.nextInt();
```

- `scnr.nextLine()` reads an input line from the standard input and copy the line into a String.
- `inSS = new Scanner(lineString);` uses the Scanner's constructor to initialize the stream's buffer to the String lineString.
- Afterwards, the program extracts input from that stream using the next() methods.

Ex. Output String Stream

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

      // Basic character stream for fullname
      StringWriter fullnameCharStream = new StringWriter();
      // Augments character stream (fullname) with print()
      PrintWriter fullnameOSS = new PrintWriter(fullnameCharStream);

      String firstName;      // First name
      String lastName;       // Last name
      String fullName;       // Full name (first and last)

      // Prompt user for input
      System.out.print("Enter \"firstname lastname age\": \n   ");
      firstName = scnr.next();
      lastName = scnr.next();
      userAge = scnr.nextInt();

      // Writes formatted string to buffer, copies from underlying char buffer
      fullnameOSS.print(lastName + ", " + firstName);
      fullName = fullnameCharStream.toString();

      // Output parsed input
      System.out.println("\n   Full name: " + fullName);
```

- An ***output string stream*** can be created that is associated with a string rather than with the screen (standard output). An output string stream is created using both the StringWriter and PrintWriter classes, which are available by including: `import java.io.StringWriter;` and `import java.io.PrintWriter;`
    - The StringWriter class provides a character stream that allows a programmer to output characters.
    - The PrintWriter class is a wrapper class that augments character streams, such as StringWriter, with print() and println() methods that allow a programmer to output various data types (e.g., int, double, String, etc.) to the underlying character stream in a manner similar to System.out.
    - To create a PrintWriter object, the program must first create a StringWriter, passing the StringWriter object to the constructor for the PrintWriter. Once the PrintWriter object is created, a program can insert characters into that stream using print() and println(). The program can then use the StringWriter's ***toString()*** method to copy that buffer to a String.
    - Notice that the PrintWriter object provides the print() and println() methods for writing to the stream, and the StringWriter object provides the toString() method for getting the resulting String.

Ex. File input

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.IOException;

public class FileReadNums {
   public static void main (String[] args) throws IOException {
      FileInputStream fileByteStream = null; // File input stream
      Scanner inFS = null;                   // Scanner object
      int fileNum1;                          // Data value from file
      int fileNum2;                          // Data value from file

      // Try to open file
      System.out.println("Opening file numFile.txt.");
      fileByteStream = new FileInputStream("numFile.txt");
      inFS = new Scanner(fileByteStream);

      // File is open and valid if we got this far
      // (otherwise exception thrown)
      // numFile.txt should contain two integers, else problems
      System.out.println("Reading two integers.");
      fileNum1 = inFS.nextInt();
      fileNum2 = inFS.nextInt();

      // Output values read from file
      System.out.println("num1: " + fileNum1);
      System.out.println("num2: " + fileNum2);
      System.out.println("num1+num2: " + (fileNum1 + fileNum2));
```

```
      // Done with file, so try to close it
      System.out.println("Closing file numFile.txt.");
      fileByteStream.close(); // close() may throw IOException if fails
   }
}
```

- The statement `fileByteStream = new FileInputStream(str);` creates a file input stream and opens the file denoted by a String variable, str, for reading. FileInputStream's constructor also allows a programmer to pass the filename as a String literal. Ex: `fileByteStream = new FileInputStream("numFile.txt");`

11.5 File output

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.IOException;

public class TextFileWriteSample {
  public static void main(String[] args) throws IOException {
     FileOutputStream fileStream = null;
     PrintWriter outFS = null;

     // Try to open file
     fileStream = new FileOutputStream("myoutfile.txt");
     outFS = new PrintWriter(fileStream);

     // Arriving here implies that the file can now be written
     // to, otherwise an exception would have been thrown.
     outFS.println("Hello");
     outFS.println("1 2 3");

     // Done with file, so try to close
     // Note that close() may throw an IOException on failure
     outFS.close();
  }
}
```

- A FileOutputStream is a class that supports writing to a file. The FileOutputStream class inherits from OutputStream.
- After declaring a variable of type FileOutputStream, a file is opened using the FileOutputStream's constructor, which can take a file name string as an argument. The constructor throws an exception if the file cannot be opened for writing.
- FileOutputStream only contains methods for writing bytes. To write strings and other common data types, a PrintWriter is commonly used. A PrintWriter has methods such as print() and println() and can be constructed from any OutputStream.

Table 11.5.1: Basic steps for opening and writing a file.

Action	Sample code
Open the file helloWorld.txt for writing	`FileOutputStream fileStream = new FileOutputStream("helloWorld.txt");`
Create a PrintWriter to write to the file	`PrintWriter outFS = new PrintWriter(fileStream);`
Write the string "Hello World!" to the file	`outFS.println("Hello World!");`
Close the file after writing all desired data	`outFS.close();`

Ex. Exception-handling construct

```
// ... means normal code
...
try {
   ...
   // If error detected
      throw objectOfExceptionType;
   ...
}
catch (exceptionType excptObj) {
   // Handle exception, e.g., print message
}
...
```

1. A ***try*** block surrounds normal code, which is exited immediately if a throw statement executes.
2. A ***throw*** statement appears within a try block; if reached, execution jumps immediately to the end of the try block. The code is written so only error situations lead to reaching a throw. The throw statement provides an object of type ***Throwable***, such as an object of type Exception or its subclasses. The statement is said to throw an exception of the particular type. A throw statement's syntax is similar to a return statement.
3. A ***catch*** clause immediately follows a try block; if the catch was reached due to an exception thrown of the catch clause's parameter type, the clause executes. The clause is said to catch the thrown exception. A catch block is called a ***handler*** because it handles an exception.

Ex. Exception handling

```
// Get user data
 System.out.print("Enter weight (in pounds): ");
 weightVal = scnr.nextInt();
```

```
            // Error checking, non-negative weight
            if (weightVal < 0) {
               throw new Exception("Invalid weight.");
            }

            System.out.print("Enter height (in inches): ");
            heightVal = scnr.nextInt();

            // Error checking, non-negative height
            if (heightVal < 0) {
               throw new Exception("Invalid height.");
            }

            // Calculate BMI and print user health info if no input error
            // Source: http://www.cdc.gov/
            bmiCalc = ((float) weightVal
                    / (float) (heightVal * heightVal)) * 703.0f;

            System.out.println("BMI: " + bmiCalc);
            System.out.println("(CDC: 18.6-24.9 normal)");
         }
         catch (Exception excpt) {
            // Prints the error message passed by throw statement
            System.out.println(excpt.getMessage());
            System.out.println("Cannot compute health info");
         }

         // Prompt user to continue/quit
         System.out.print("\nEnter any key ('q' to quit): ");
         quitCmd = scnr.next().charAt(0);
      }
   }
}
```

PRINTS:

```
Enter weight (in pounds): 150
Enter height (in inches): 66
BMI: 24.208
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): a
Enter weight (in pounds): -1
Invalid weight.
Cannot compute health info.

Enter any key ('q' to quit): a
Enter weight (in pounds): 150
Enter height (in inches): -1
Invalid height.
Cannot compute health info.

Enter any key ('q' to quit): q
```

- The object thrown and caught must be of the Throwable class type, or a class inheriting from Throwable, ex. Error, Exception,
- The Exception class (and other Throwable types) has a constructor that can be passed a String, as in `throw new Exception("Invalid weight.");`, which allocates a new Exception object and sets an internal String value that can later be retrieved using the getMessage() method, as in `System.out.println(excpt.getMessage());`.

- A method without try/catch functionality must alternatively specify that a contained instruction may throw an exception. If a method throws an exception not handled within the method, a programmer must include a ***throws clause*** within the method declaration, by appending `throws Exception` before the opening curly brace.

```
public static int getHeight(Scanner scnr) throws Exception {
}
```

- Different throws in a try block may throw different exception types. Multiple handlers may exist, each handling a different type. The first matching handler executes; remaining handlers are skipped.
    - ***catch(Throwable thrwObj)*** is a catch-all handler that catches any error or exception as both are derived from the Throwable class; this handler is useful when listed as the last handler.

```
// ... means normal code
...
try {
   ...
   throw objOfExcptType1;
   ...
   throw objOfExcptType2;
   ...
   throw objOfExcptType3;
   ...
}
catch (ExcptType1 excptObj) {
   // Handle type1
}
catch (ExcptType2 excptObj) {
   // Handle type2
}
catch (Throwable thrwObj) {
   // Handle others (e.g., type3)
}
```

```
... // Execution continues here
```

- A programmer must ensure that files are closed when no longer in use, to allow the JVM to clean up any resources associated with the file streams.

Finally block.

```
// ... means normal code
...
try {
   ...
   // If error detected
      throw objOfExcptType;
   ...
}
catch (excptType excptObj) {
   // Handle exception, e.g., print message
}
finally {
   // Clean up resources, e.g., close file
}
...
```

- A ***finally block*** follows all catch blocks, and executes after the program exits the corresponding try or catch blocks.

```
import java.util.Scanner;
import java.io.FileWriter;
import java.io.IOException;

public class FileWriteChars {
   /* Method closes a FileWriter.
      Prints exception message if closing fails. */
   public static void closeFileWriter(FileWriter fileWriter) {
      try {
         if (fileWriter != null) { // Ensure fileWriter references a valid object
            System.out.println("Closing file.");
            fileWriter.close(); // close() may throw IOException if fails
         }
      } catch (IOException closeExcpt) {
         System.out.println("Error closing file: " + closeExcpt.getMessage());
      }
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      final int NUM_CHARS_TO_WRITE = 10; // Num chars to write to file
      int countVar;                      // Track num chars written so far
      FileWriter fileWriter = null;      // FileWriter for writing file
      String fileName;                   // User defined file name
      char charWrite;                    // Char data written to file
```

```
      countVar = 0;
      charWrite = 'a';

      // Get file name from user
      System.out.print("Enter a valid file name: ");
      fileName = scnr.next();

      try {
         System.out.println("Creating file " + fileName + ".");
         fileWriter = new FileWriter(fileName); // May throw IOException

         // Use file output stream
         System.out.print("Writing " + NUM_CHARS_TO_WRITE + " characters: ");
         while (countVar < NUM_CHARS_TO_WRITE) {
            fileWriter.write(charWrite);
            System.out.print(charWrite);

            charWrite++; // Get next char ready
            countVar++;  // Keep track of number chars written
         }
         System.out.println();
      } catch (IOException excpt) {
         System.out.println("Caught IOException: " + excpt.getMessage());
      } finally {
         closeFileWriter(fileWriter); // Ensure file is closed!
      }
   }
}
```

Chapter 12 - Inheritance

A derived class (or subclass) is a class that is derived from another class, called a base class (or superclass). Any class may serve as a base class. The derived class is said to inherit the properties of the base class, a concept called inheritance. An object declared of a derived class type has access to all the public members of the derived class as well as the public members of the base class.
A derived class is declared by placing the keyword extends after the derived class name, followed by the base class name. Ex: class DerivedClass extends BaseClass { ... }. The figure below defines the base class GenericItem and derived class ProduceItem that inherits from GenericItem.

Class ProduceItem is derived from class GenericItem.

GenericItem.java

```
public class GenericItem {
   private String itemName;
   private int itemQuantity;
```

```
    public void setName(String newName) {
        itemName = newName;
    }

    public void setQuantity(int newQty) {
        itemQuantity = newQty;
    }

    public void printItem() {
        System.out.println(itemName + " " + itemQuantity);
    }
}
```

ProduceItem.java

```
public class ProduceItem extends GenericItem {
    private String expirationDate;

    public void setExpiration(String newDate) {
        expirationDate = newDate;
    }

    public String getExpiration() {
        return expirationDate;
    }
}
```

Various inheritance variations are possible:

- A derived class can serve as a base class for another class. Ex: `class FruitItem extends ProduceItem {...}` creates a derived class FruitItem from ProduceItem, which was derived from GenericItem.
- A class can serve as a base class for multiple derived classes. Ex: `class FrozenFoodItem extends GenericItem {...}` creates a derived class FrozenFoodItem that inherits from GenericItem, just as ProduceItem inherits from GenericItem.

A class can only be derived from one base class directly. Ex: Inheriting from two classes as in `class House extends Dwelling, Property {...}` results in a compiler error.

The members of a derived class have access to the public members of the base class, but not to the private members of the base class. This is logical—allowing access to all private members of a class merely by creating a derived class would circumvent the idea of private members. Thus, adding the following member method to the Restaurant class yields a compiler error.

Recall that members of a class may have their access specified as *public* or *private*. A third access specifier is ***protected***, which provides access to derived classes and all classes in the same ***package*** but not by anyone else. Packages are discussed in detail elsewhere, but for our purposes a package can just be thought of as the directory in which program files are located. Thus, classes in the same package are located in the same directory. The following illustrates the implications of the protected access specifier.

In the following example, the member called name is specified as protected and is accessible anywhere in the derived class. Note that the name member is also accessible in main()—the protected specifier also allows access to classes in the same package; protected members are private to everyone else.

```
public class Business{

   protected String name;    // Member accessible by self and derived classes

   private String address;   // Member accessible only by self


   public void printMembers() { // Member accessible by anyone

      // Print information ...

   }


}
```

Separately, the keyword "public" in a class definition like `public class DerivedClass {...}` specifies a class's visibility in other classes in the program:

- *public* : A class can be used by every class in the program regardless of the package in which either is defined.
- *no specifier* : A class can be used only in other classes within the same package, known as ***package-private***.

When a derived class defines a member method that has the same name and parameters as a base class's method, the member method is said to ***override*** the base class's method. The example below shows how the Restaurant's getDescription() method overrides the Business's getDescription() method.

The ***@Override*** annotation is placed above a method that overrides a base class method so the compiler verifies that an identical base class method exists. An ***annotation*** is an optional command beginning with the "@" symbol that can provide the compiler with information that helps the compiler detect errors better. The @Override annotation causes the compiler to produce an error when a programmer mistakenly specifies parameters that are different from the parameters of the method that should be overridden or misnames the overriding method. Good practice is to always include an @Override annotation with a method that is meant to override a base class method.

An overriding method can call the overridden method by using the super keyword. Ex: `super.getDescription()`. The ***super*** keyword is a reference variable used to call the parent class's methods or constructors.

The Object class

The built-in ***Object class*** serves as the base class for all other classes and does not have a base class. All classes, including user-defined classes, are derived from Object and implement Object's methods. In the following discussion, note the subtle distinction between the term "Object class" and the generic term "object", which can refer to the instance of any class. Two common methods defined within the Object class are toString() and equals().

- The ***toString()*** method returns a String representation of the Object. By default, toString() returns a String containing the object's class name followed by the object's hexadecimal address in memory. Ex: java.lang.Object@372f7a8d.
- The ***equals(otherObject)*** method compares an Object to otherObject and returns true if both variables reference the same object. Otherwise, equals() returns false. By default, equals() tests the equality of the two Object references, not the equality of the Objects' contents.

Polymorphism refers to determining which program behavior to execute depending on data types. Method overloading is a form of ***compile-time polymorphism*** wherein the compiler determines which of several identically-named methods to call based on the method's arguments. Another form is ***runtime polymorphism*** wherein the compiler cannot make the determination but instead the determination is made while the program is running.

One scenario requiring runtime polymorphism involves derived classes. Programmers commonly create a collection of objects of both base and derived class types. Ex: the statement `ArrayList<GenericItem> inventoryList = new ArrayList<GenericItem>();` declares an ArrayList that can contain references to objects of type GenericItem or ProduceItem. ProduceItem derives from GenericItem.

The concept of inheritance is commonly confused with the idea of composition. Composition is the idea that one object may be made up of other objects, such as a MotherInfo class being made up of objects like firstName (which may be a String object), childrenData (which may be an ArrayList of ChildInfo objects), etc. Defining that MotherInfo class does *not* involve inheritance, but rather just composing the sub-objects in the class.

The 'has-a' relationship. A MotherInfo object 'has a' String object and 'has a' ArrayList of ChildInfo objects, but no inheritance is involved.

```
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;

   ...
}
```

In contrast, a programmer may note that a mother is a kind of person, and all persons have a name and birthdate. So the programmer may decide to better organize the program by defining a PersonInfo class, and then by creating the MotherInfo class derived from PersonInfo, and likewise for the ChildInfo class.

The 'is-a' relationship. A MotherInfo object 'is a' kind of PersonInfo. The MotherInfo class thus inherits from the PersonInfo class. Likewise for the ChildInfo class.

Inheritance

```
public class PersonInfo {
   public String firstName;
   public String birthdate;

   ...
}

public class ChildInfo extends PersonInfo {
   public String schoolName;

   ...
}

public class MotherInfo extends PersonInfo {
   public String spousename;
   public ArrayList<ChildInfo> childrenData;
  ...
}
```

Because all classes are derived from the Object class, programmers can take advantage of runtime polymorphism in order to create a collection (e.g., ArrayList) of objects of various class types and perform operations on the elements. The program below uses the Business class and other built-in classes to create and output a single ArrayList of differing types.
Note that a method operating on a collection of Object elements may only invoke the methods defined by the base class (e.g., the Object class). Thus, a statement that calls the toString() method on an element of an ArrayList of Objects called objList, such as

`objList.get(i).toString()`, is valid because the Object class defines the toString() method. However, a statement that calls, for example, the Integer class's intValue() method on the same element (i.e., `objList.get(i).intValue()`) results in a compiler error even if that particular element is an Integer object.

Projects

Project 3 - Gradebook (2 ways)

//Store grades, compute final grade and letter grade
public class Gradebook {

//attribute list

```
private double projectAvg;
private double labAvg;
private double zybooksAvg;
private double exam1;
private double exam2;
private double exam3;
private double exam4;
```

//method list (getters and setters)

```
    public void setProjectAvg(double projectAverage) {
  projectAvg = projectAverage;
}
public double getProjectAvg(){
  return (projectAvg);
              }
public void setLabAvg(double labAverage) {
  labAvg = labAverage;
}
public double getlabAvg(){
  return (labAvg);
              }
public void setZybooksAvg(double zybooksAverage) {
  zybooksAvg = zybooksAverage;
}
public double getZybooksAvg(){
  return (zybooksAvg);
              }
public void setExam1(double exam1Grade) {
  exam1 = exam1Grade;
}
public double getExam1(){
  return (exam1);
              }
public void setExam2(double exam2Grade) {
  exam2 = exam2Grade;
}
public double getExam2(){
  return (exam2);
              }
public void setExam3(double exam3Grade) {
  exam3 = exam3Grade;
}
public double getExam3(){
  return (exam3);
```

```
                }
    public void setExam4(double exam4Grade) {
      exam4 = exam4Grade;
    }
    public double getExam4(){
      return (exam4);
                }

}
```

```
import java.util.Scanner;

public class GradebookTest{

  public static void main(String[] args){

  Scanner scnr = new Scanner(System.in);
  Gradebook grades = new Gradebook();

  //taking user input for fields
  double projectAvgGrade = 0;
  System.out.print("Enter project average: ");
  projectAvgGrade = scnr.nextDouble();

  double labAvgGrade = 0;
  System.out.print("Enter lab average: ");
  labAvgGrade = scnr.nextDouble();

  double zybooksAvgGrade = 0;
  System.out.print("Enter Zybooks average: ");
  zybooksAvgGrade = scnr.nextDouble();

  //array that takes user input for exam grades
  final int NUM_EXAMS_MAX = 4;
  double[] examGrades = new double[NUM_EXAMS_MAX];
    int i;
    int examNumber = 0;
    for (i = 0; i < examGrades.length; ++i) {
      examNumber += 1;
      System.out.print("Enter Exam " + examNumber + " grade: ");
      examGrades[i] = scnr.nextDouble();
    }

  //calling setters to store user input in private fields
  grades.setProjectAvg(projectAvgGrade);
  grades.setLabAvg(labAvgGrade);
  grades.setZybooksAvg(zybooksAvgGrade);
  grades.setExam1(examGrades[0]);
  grades.setExam2(examGrades[1]);
  grades.setExam3(examGrades[2]);
  grades.setExam4(examGrades[3]);

//print user-inputted exam grades
double examTotalPoints = 0;
int numberOfExams = 0;
System.out.println("Your exam grades are " + grades.getExam1() + ", " + grades.getExam2() + ", " + grades.getExam3() + ", " + grades.getExam4() + ".");

//ask user if they want to drop each exam
System.out.print("Would you like to drop exam 1? Type \"yes\" or \"no\": ");
  String yesOrNo = scnr.next();
if (yesOrNo.equals("no")){
  examTotalPoints += grades.getExam1();
  numberOfExams += 1;
}
System.out.print("Would you like to drop exam 2? Type \"yes\" or \"no\": ");
  String yesOrNo1 = scnr.next();
if (yesOrNo1.equals("no")){
```

```
  examTotalPoints += grades.getExam2();
  numberOfExams += 1;
}
System.out.print("Would you like to drop exam 3? Type \"yes\" or \"no\": ");
  String yesOrNo2 = scnr.next();
if (yesOrNo2.equals("no")){
  examTotalPoints += grades.getExam3();
  numberOfExams += 1;
}
System.out.print("Would you like to drop exam 4? Type \"yes\" or \"no\": ");
  String yesOrNo3 = scnr.next();
if (yesOrNo3.equals("no")){
  examTotalPoints += grades.getExam4();
  numberOfExams += 1;
}
if (yesOrNo.equals("yes") && yesOrNo1.equals("yes") && yesOrNo1.equals("yes") && yesOrNo1.equals("yes")){
  System.out.print("You cannot drop all exam grades.");
  numberOfExams = -1;
}
```

//calculates and prints exam average, and if it's at least 50

```
double examAverage = examTotalPoints / numberOfExams;
System.out.print("Your exam average is " + examAverage + ". ");

if(examAverage >= 50){
  System.out.println("It is at least 50.");
}
else{
  System.out.println("It is less than 50.");
}
```

//asks user to input weights

```
System.out.print("What percent of your final grade are projects? Enter the number without the % symbol: ");
  double weightProject = scnr.nextDouble();
System.out.print("What percent of your final grade are labs? Enter the number without the % symbol: ");
  double weightLab = scnr.nextDouble();
System.out.print("What percent of your final grade is Zybooks? Enter the number without the % symbol: ");
  double weightZybooks = scnr.nextDouble();
System.out.print("What percent of your final grade are exams? Enter the number without the % symbol: ");
  double weightExam = scnr.nextDouble();
```

//checks if weights equal 100%, calculates final grade, and gives letter grade

```
double finalGrade = (weightProject * projectAvgGrade) + (weightLab * labAvgGrade) + (weightZybooks * zybooksAvgGrade) + (weightExam * examAverage);

if (weightProject + weightLab + weightZybooks + weightExam != 100){
  System.out.print("-1.0 (weight totals do not equal 100.0%)");
}

else{
  finalGrade = finalGrade / 100;

  System.out.println("Your final grade is " + finalGrade + ".");
  if (finalGrade < 60 || examAverage < 50){
    System.out.print("Your final letter grade is F.");
  }
  else if (finalGrade < 70){
    System.out.print("Your final letter grade is D.");
  }
  else if (finalGrade < 80){
    System.out.print("Your final letter grade is C.");
  }
  else if (finalGrade < 90){
    System.out.print("Your final letter grade is B.");
  }
  else if (finalGrade >= 90){
    System.out.print("Your final letter grade is A.");
  }
}



  }
}
```

//alternate way to solve project (takes test cases using constructor instead of user input)

```
public class Gradebook {
```

//attribute list

```
    private double projectAvg;
    private double labAvg;
    private double zybooksAvg;
    private double exam1;
    private boolean exam1Drop;
    private double exam2;
    private boolean exam2Drop;
    private double exam3;
    private boolean exam3Drop;
    private double exam4;
    private boolean exam4Drop;
    private double weightOfProject;
    private double weightOfLab;
    private double weightOfZybooks;
    private double weightOfExam;
```

//constructor (initializes all fields)

```
public Gradebook(double projectAverage, double labAverage, double zybooksAverage,
double exam1Average, boolean dropExam1, double exam2Average, boolean dropExam2,
double exam3Average, boolean dropExam3, double exam4Average, boolean dropExam4,
double weightProject, double weightLab, double weightZybooks, double weightExam){
   projectAvg = projectAverage;
   labAvg = labAverage;
   zybooksAvg = zybooksAverage;

   //if dropExam boolean is true, the exam is dropped
   exam1 = exam1Average;
   exam1Drop = dropExam1;
   exam2 = exam2Average;
   exam2Drop = dropExam2;
   exam3 = exam3Average;
   exam3Drop = dropExam3;
   exam4 = exam4Average;
   exam4Drop = dropExam4;

   weightOfProject = weightProject;
   weightOfLab = weightLab;
   weightOfZybooks = weightZybooks;
   weightOfExam = weightExam;
}
```

//method list (getters and setters)

```
 public void setProjectAvg(double projectAverage) {
   projectAvg = projectAverage;
}
public double getProjectAvg(){
   return (projectAvg);
                }
public void setLabAvg(double labAverage) {
   labAvg = labAverage;
}
public double getLabAvg(){
   return (labAvg);
                }
public void setZybooksAvg(double zybooksAverage) {
   zybooksAvg = zybooksAverage;
}
public double getZybooksAvg(){
   return (zybooksAvg);
                }
public void setExam1(double exam1Grade) {
   exam1 = exam1Grade;
}
public double getExam1(){
   return (exam1);
                }
public void setExam2(double exam2Grade) {
   exam2 = exam2Grade;
```

```
}
public double getExam2(){
  return (exam2);
            }
public void setExam3(double exam3Grade) {
  exam3 = exam3Grade;
}
public double getExam3(){
  return (exam3);
            }
public void setExam4(double exam4Grade) {
  exam4 = exam4Grade;
}
public double getExam4(){
  return (exam4);
            }
```

//Method that calculates all grades

```
   public void calculateGrades() {

  //calculates exam average
  double examTotalPoints = 0;
  int numberOfExams = 0;

  if (exam1Drop == false){
    numberOfExams += 1;
    examTotalPoints += exam1;
  }
  if (exam2Drop == false){
    numberOfExams += 1;
    examTotalPoints += exam2;
  }
  if (exam3Drop == false){
    numberOfExams += 1;
    examTotalPoints += exam3;
  }
  if (exam4Drop == false){
    numberOfExams += 1;
    examTotalPoints += exam4;
  }

  if (exam1Drop == true && exam2Drop == true && exam3Drop == true && exam4Drop == true){
    examTotalPoints = 0;
    numberOfExams = 1;        //prevents division by 0 if all exam grades are dropped
  }

  double examAverage = examTotalPoints / numberOfExams;
      System.out.print("Your exam average is " + examAverage + ". " );
```

//Determines if exam average is at least 50

```
       if (examAverage >= 50.0){
   System.out.println("It is at least 50.");
}
else{
   System.out.println("It is below 50.");
}
```

//Calculates final grade and letter grade

```
      double finalGrade = (weightOfProject * projectAvg) + (weightOfLab * labAvg) + (weightOfZybooks * zybooksAvg) + (weightOfExam * examAverage);

  String letterGrade = "z";   //initiated to impossible grade--will be changed by statements below

  if (weightOfProject + weightOfLab + weightOfZybooks + weightOfExam != 100){
     finalGrade = -1;      //returns grade of -1 if grade weights don't add to 100%
     letterGrade = "N/A (grade weights don't equal 100%)";
  }
  else{
     finalGrade = finalGrade / 100;

     if (finalGrade < 60 || examAverage < 50){
        letterGrade = "F";
     }
     else if (finalGrade < 70){
        letterGrade = "D";
```

```
            }
            else if (finalGrade < 80){
                letterGrade = "C";
            }
            else if (finalGrade < 90){
                letterGrade = "B";
            }
            else if (finalGrade >= 90){
                letterGrade = "A";
            }

        }

        System.out.println("Your final grade is " + finalGrade + " and your letter grade is " + letterGrade + ".");
    }
}
```

//test file (has one example, but should need more)

```
public class GradebookTest{

  public static void main(String[] args){
```

//check that numbers are stored and calculations run
//check failing final grade
//test getter methods

```
        Gradebook grades1 = new Gradebook(1, 2, 3, 4, false, 5, false, 6, false, 7, false, 25, 25, 25, 25);

        System.out.println("Your project average grade is " + grades1.getProjectAvg());
        System.out.println("Your lab average grade is " + grades1.getLabAvg());
        System.out.println("Your Zybooks average grade is " + grades1.getZybooksAvg());
        System.out.println("Your exam grades are " + grades1.getExam1() + ", " + grades1.getExam2() + ", " + grades1.getExam3() + ", " + grades1.getExam4() + ".");

        grades1.calculateGrades();
        System.out.println("");
```

Project 4 - Fish, Flush, Fuss with Javadoc Comments

```
/**
* A class representing solutions to the children's game Fish Flush Fuss.
* @author Anna Li
* @version 1.0
*/


public class FishFlushFuss {
```

//attribute list

```
    /**
   * Fish value
   */

    private int Fish;


   /**
   * Flush value
   */


    private int Flush;


   /**
   * Fuss value
   */
```

```
 private int Fuss;
```

//constructor

```
 /**
* Constructor initializing Fish to fish, Flush to flush, and Fuss to fuss.
* @param fish the fish value
* @param flush the flush value
* @param fuss the fuss value
*/

public FishFlushFuss(int fish, int flush, int fuss) {
   Fish = fish;
   Flush = flush;
   Fuss = fuss;
}
```

//method that calculates current round

```
 /**
* Gives the answer to the current round of the game
* @param currentRound integer representing the current round of play
* @return String representing the answer to the current round
*/

public String playRound(int currentRound){
   String roundAnswerString = "";
   int roundAnswerNumber;

   //condition if round is multiple of all three values
   if (currentRound % Fish == 0 && currentRound % Flush == 0 && currentRound % Fuss == 0){
      roundAnswerString = "Flamingo";
   }

   //condition if round is divisible by Fish value
   else if (currentRound % Fish == 0) {
      if (currentRound % Flush == 0) {         //checks if round is also divisible by Flush value
          if (Fish > Flush) {
              roundAnswerString = "Fish";
          }
          else {
              roundAnswerString = "Flush";
          }
      }
      else if (currentRound % Fuss == 0) {       //checks if round is also divisible by Fuss value
         if (Fish > Fuss) {
            roundAnswerString = "Fish";
         }
         else {
            roundAnswerString = "Fuss";
         }
      }
      else {
         roundAnswerString = "Fish";              //result if round is only divisible by Fish value
      }
   }


   //condition if round is divisible by Flush value, but not Fish value
   else if (currentRound % Flush == 0) {
      if (currentRound % Fuss == 0) {             //checks if round is also divisible by Fuss value
         if (Flush > Fuss) {
            roundAnswerString = "Flush";
         }
         else {
            roundAnswerString = "Fuss";
         }
      }
      else {
         roundAnswerString = "Flush";             //result if round is only divisible by Flush value
      }
   }

   //condition if round is multiple of Fuss value only
   else if (currentRound % Fuss == 0){
      roundAnswerString = "Fuss";
   }
```

```
    //condition if round is not divisible by any of the 3 values
    else {
      roundAnswerNumber = currentRound;
      roundAnswerString = "" + roundAnswerNumber;              //compiler treats integers as String after concatenation
    }
    return roundAnswerString;
  }
```

// gives result of entire game

```
  /**
  * Gives result of the entire game
  * @param numRounds the number of rounds the game runs for
  * @return String representing the game's entire result
  */

  public String playGame (int numRounds) {
    String entireGame = "";
    for (int currentRound = 1; currentRound <= numRounds; ++currentRound) {
      entireGame += "Round " + currentRound + ": " + playRound(currentRound) + "\n";
    }
    return entireGame;
  }
```

//getters

```
   /**
  * Returns value of Fish
  * @return int representing value of Fish
  */

  public int getFish(){
    return Fish;
                }

  /**
  * Returns value of Flush
  * @return int representing value of Flush
  */

  public int getFlush(){
    return Flush;
                }

  /**
  * Returns value of Fuss
  * @return int representing value of Fuss
  */

  public int getFuss(){
    return Fuss;
                }


}
```

//main method file that I did not write

```
import java.util.*;

public class FishFlushFussMain{

  public static void main(String[] args){

    // The scanner we will be using to accept user input
    Scanner scan = new Scanner(System.in);
   /*  Initializing fish, flush, fuss, and round to invalid values to cause them
    *  to enter their respective User Input loops.
    */
    int fish = -1;
    int flush = -1;
    int fuss = -1;
```

```
    int rounds = -1;

    // Our FishFlushFuss object
    FishFlushFuss scp;

    // Capture user input for fish, while also ensuring correctness
    System.out.println("Enter an integer for your fish number.");
    while(fish < 0){
      System.out.println("Please enter a number greater than zero:");
      try{
        fish = Integer.parseInt(scan.nextLine());
      } catch (Exception e) {
        System.out.println("That is not a valid number. Please enter an integer greater than zero.");
      }
    }

    // Capture user input for flush, while also ensuring correctness
    System.out.println("Enter an integer for your flush number.");
    while(flush < 0){
      System.out.println("Please enter a number greater than zero that is not equal to the fish number:");
      try{
        flush = Integer.parseInt(scan.nextLine());
        if(flush == fish) {
          System.out.println("Input invalid. Make sure that this number is not equal to the fish number.");
          flush = -1;
        }
      } catch(Exception e) {
        System.out.println("That is not a valid number. Please enter an integer greater than zero.");
      }
    }

    // Capture user input for fuss, while also ensuring correctness
    System.out.println("Enter an integer for your fuss number.");
    while(fuss < 0){
      System.out.println("Please enter a number greater than zero that is not equal to the fish or flush number:");
      try{
        fuss = Integer.parseInt(scan.nextLine());
        if(fuss == fish || fuss == flush) {
          System.out.println("Input invalid. Make sure that this number is not equal to the fish or flush number.");
          fuss = -1;
        }
      } catch(Exception e) {
        System.out.println("That is not a valid number. Please enter an integer greater than zero.");
      }
    }

    // Initializing our FishFlushFuss object with the values we just obtained from the user
    scp = new FishFlushFuss(fish, flush, fuss);

    // Capture user input for rounds, while also ensuring correctness
    System.out.println("How many rounds would you like to play?");
    while(rounds < 0){
      System.out.println("Please enter a number greater than zero:");
      try{
        rounds = Integer.parseInt(scan.nextLine());
      } catch(Exception e) {
        System.out.println("That is not a valid number. Please enter an integer greater than zero.");
      }
    }

    // Printing results
    System.out.println("Results:");
    System.out.println(scp.playGame(rounds));

  }

}
```

Project 5 - Detect Total Orders

```
public class DetectTotalOrder {

  private int[][] arrayOfRelations;
  private int dimension;
```

//Constructor

```
public DetectTotalOrder(int[][] matrix, int matrixDimensions) {
  arrayOfRelations = matrix;
  dimension = matrixDimensions;
}
```

//Method List

//checks if for every case where A is related to B and B is related to C, A is related to C

```
 //returns true for transitive, false for not transitive
public boolean detectTransitive() {

  for (int i = 0; i < dimension; ++i) {          //loop for every element
    for(int j = 0; j < dimension; ++j) {
      if (arrayOfRelations[i][j] == 1) {              //if element at index (i,j) is 1,
        for(int k = 0; k < dimension; ++k){          //then check if j has relation to anything
          if (arrayOfRelations[j][k] == 1){
            if (arrayOfRelations[i][k] == 0){
              System.out.println("The relation is not transitive.");
              return false;
            }
          }
        }
      }
    }
  }
  System.out.println("The relation is transitive.");
  return true;  //true if outer loop finishes
}
```

//checks if every element is related to itself

```
//returns true for reflexive, false for not reflexive
public boolean detectReflexive() {

  for (int i = 0; i < dimension; ++i) {
    for(int j = 0; j < dimension; ++j) {
      if (arrayOfRelations[i][i] == 0) {
        System.out.println("The relation is not reflexive.");
        return false;
      }
    }
  }
  System.out.println("The relation is reflexive.");
  return true;
}
```

//checks if the only cases where we have xRy and yRx is when x==y

```
//returns true for antisymmetric, false for not antisymmetric
public boolean detectAntisymmetric() {

  for (int i = 0; i < dimension; ++i) {                  //loop for every element
    for(int j = 0; j < dimension; ++j) {
      if (arrayOfRelations[i][j] == 1){                      //if element at index (i,j) is 1,
        if (i != j){                                  //checks that element is not at index (i,i)
          if (arrayOfRelations[j][i] == 1){
            System.out.println("The relation is not antisymmetric.");
            return false;
          }
        }
      }
    }
  }
  System.out.println("The relation is antisymmetric.");
  return true;
}
```

```
    //checks if a relation is transitive, reflexive, and antisymmetric
  public void detectTotalOrder() {

    System.out.println("");
    if (detectTransitive() && detectReflexive() && detectAntisymmetric()){
      System.out.println("Therefore, this two-dimensional matrix defines a total order."+ "\n");
    }
    else {
      System.out.println("Therefore for at least this reason, this two-dimensional matrix does not define a total order." + "\n");
    }
  }

}
```

//scrapped user input method

```
import java.util.Scanner;

public class DetectTotalOrdersTest{
  public static void main(String[] args){

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

//allows user to decide size of matrix

```
    System.out.print("Enter the number of elements you wish to detect relations for: ");
  int numElem = scnr.nextInt();

  int[][] arrayOfRelations = new int[numElem][numElem];
```

//takes user input for matrix

```
    System.out.println("Enter rows of relations with a space between each element (1 for yes, 0
  for no). " + "\n" + "For example, if item A is related to itself and to item B, but not to item C,
  enter \"1 1 0\"");

 for (int i = 0; i < numElem; ++i) {
    System.out.print("Enter row (1 for yes, 0 for no): ");
    for(int j = 0; j < numElem; ++j) {
      arrayOfRelations[i][j] = scnr.nextInt();
    }
  }
```

//prints the entered matrix

```
    System.out.println("The matrix you entered: ");
    for(int[] x : arrayOfRelations){
      for(int y : x) {
        System.out.print(y + " ");
      }
      System.out.println();
    }
  }
}
```

Labs

Lab 8

Complete the recursive method that prints a given word in reverse.

Answer:

```
public static void reverseWord(String s) {
```

```
public static void reverseWord(String s) {
   //write the base case
     if(s.length() == 1){
        System.out.print(s.charAt(0));
     }
   //write the recursive case to print the word
     if (s.length() > 1){
        System.out.print(s.charAt(s.length() - 1));
        String substring = s.substring(0, s.length() - 1);
        s = substring;
        reverseWord(s);
     }
}
```

Complete the recursive method that returns the exponent of a given number.

Answer:

```
public static int raiseToPower(int base, int exponent) {
     //write the base case
     if(exponent == 0) {
          return 1;
     }



     //write the recursive case
     else {
          return base * raiseToPower(base, exponent - 1);

     }


   }
```

Complete the recursive method that returns the factorial of a given number.

Answer:

```
public static int factorial(int number) {

      //write the base case
```

```
if(number == 0) {
 return 1;
}
else if (number < 0){
 return -1;
}

      //write the recursive case
else {
 return number * factorial(number - 1);
}



   }
```

Given the following code:

```
public static int function(int y) {
     if (y == 1)
          return 5;
     else {
          function(y - 1);
          y = y + 1;
          return 83;
     }
}
```

What value is returned when `function(2)` is executed?

A. 83 //no return so the "5" stays in the called method

B. 5

~~**C. 6**~~

D. Infinite recursion