

Lesson 1: Installing Java

- The JDK contains the compiler and their tools needed to create executable Java programs
- The compiler converts the code you write into bytecode that the Java Virtual Machine can read
- The main difference between OracleJDK and OpenJDK is the licensing and what you can do with the software created
- The JVM is a software layer between the java code and the machine running the software. It acts as a translator between our code and the machine's code, allowing it to run on many different devices
- Before Java 9, you had to install a compatible Java Runtime Environment (JRE), which included the required JVM to run software but not the tools required to develop software
- A Java class is a basic container for code, as is a method
 - The main method is the entry point for every Java program

USING THE COMPILER AND JAVA VIRTUAL MACHINE

The flow chart in Figure 1.21 represents the steps that happen behind the scenes when we compile and run our program.

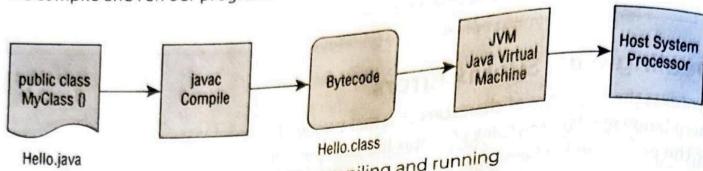


Figure 1.21 Behind the scenes while compiling and running

1. We created the Hello.java file using normal text and Java syntax. It includes a public class that defines what we want the computer to do when we run the program.
2. When we ran the command `javac Hello.java` in the second step, it invoked the Java compiler. The compiler converts the text that we typed in Hello.java into bytecode.
3. Bytecode is a set of instructions for the JVM. When we ran the command `java Hello`, we were running the bytecode version of the program that the JVM can read and interpret.
4. The JVM is basically a virtual computer that runs in a layer between the software and the operating system. Because all Java programs use the JVM to run, we can run the same program on any platform that supports Java, including Windows, macOS, and Linux.
5. The JVM interprets the bytecode from your program and converts it into machine code that can run on the processor of your machine.

SUMMARY

In this lesson, we went over what a JDK is and why we need one as Java developers. We compared Oracle JDK with OpenJDK and then installed OpenJDK for use in this book.

Lesson 2: The IDE

- The 3 biggest features of an IDE are the source code editor, build automation tools, and the debugger
 - The source code editor is where you type code. It usually provides tools like auto-alignment, auto-complete, syntax error highlighting, etc.
 - The build automation tools let you compile and run your code, along with integrating with external tools, all with the click of a button
 - The debugger lets you freeze your code in time and look at exactly what a program is doing at that moment, so you can figure out why your code isn't doing what it's supposed to be doing

Lesson 3: Using an IDE

- Maven is a build management system for Java projects. External build systems make it easy for everyone to get their jobs done without relying on a particular IDE
- The Group Id value identifies who owns and maintains the code, similar to how a domain name works in a web address. Use something that identifies you as the developer, like initials
- Version indicates the project version and can integrate with Git
- Artifact Id and Package are okay to leave on default
- The Build button completely recompiles all code in the program. The Clean and Rebuild button identifies code that has changed since the last build and recompiles only the classes that contain the changed code, which is more efficient in larger programs
- Exceptions are errors that prevent the code from running after it has been compiled; they include things like dividing a number by 0
- Logic errors are related to how we define the values that go into the code and produce unexpected output (i.e. $5 * 6$ results in 35 instead of 30)

Lesson 4: Using Program Statements and Variables

- Identifiers
 - An identifier is a sequence of characters used to name a variable, method, class or package.
 - It cannot span multiple lines
 - Must only contain numbers, letters, underscores, dashes, and dollar signs
 - Cannot contain any spaces
 - Can not be an underscore on its own
 - **p. 79 reserved keywords**
 - Java Data Types
 - An internal representation of a piece of data and the operations that can be applied to that data
 - Two kinds: primitive (part of the language definitions) and reference (developer-defined)
 - **Table 4.2**
 - Java is a statically typed language, which means we must say the type of a piece of data before we use it. All types of data in our program are

known at compile time (declaring in the program will let the compiler know what the data piece is before we use it)

- Type Conversion
 - The compiler will automatically convert a narrower data type (with fewer bits) to a wider type (with a larger number of bits)
 - In the reverse, the developer must explicitly tell the compiler how to do this. Think of fitting a carry-on item into a large luggage, as opposed to the opposite
 - The ability of the compiler to check that all types are compatible is known as type safety, which prevents you from doing things undefined or unexpected
- Literals
 - A sequence of characters that represents a data item in the source code
 - boolean: reserved words true and false
 - char: represented by a single character surrounded by single quotes ('B'), a Unicode espace sequence (u\100), or regular escape sequence (\n)
 - floating-point number: examples are 3.14, 2.56E31, 4.56D. The letter F or f can be used to specify a float data type; D or d is used to specify a double data type
 - Whole numbers: Integer literals can be written as a decimal (104) or hexadecimal (0x19F)
 - Null: The reserved word null is a null literal
 - String: Sequence of characters surrounded by double quotes
- Variables
 - Declared as data_type variable_name;
 - Examples:
 - int counter
 - counter=7;
 - int counter = 7;
- Constants are denoted with the final keyword; variable names should be in all caps. Example: final int MAX_AGE = 99;
- Operators
 - Three flavors
 - Unary Operators
 - Require only one operand
 - Can be prefix (coming before the operand) or postfix (coming after the operand)
 - **Table 4.3**
 - Binary Operators
 - Require two operands
 - Are infix (between the operands)
 - Include comparison operators
 -

- **Table 4.4**
- Ternary Operators
 - Require 3 operands
 - ?:
- Operator Precedence **Table 4.5**

Lesson 5: Collecting and Parsing Input from the User

- A Scanner object is what allows you to read information from the standard input, which is generally the console. It is a non-primitive data type
- Everything that comes in on the console command line is text, which must be stored in strings
- `myScanner.nextLine()` waits for the user to type something into the console and hit the Enter key
- We can use `parseInt` and `parseFloat` to convert Strings to integers and floating numbers, respectfully, but only if we can fully trust the input, otherwise we'll get an error
- Why not use `nextInt()` etc.?
 - `nextLine()` sits and waits until the user hits Enter. When the user hits Enter, then the program knows to read the information that was typed into the console command line. In other words, it reads the entire line including the typed-in return
 - However, `nextInt()` only reads the number that was entered, and it ignores the carriage return that the user entered. Also throws an error if anything other than a number is entered
 - In other words, if you hit enter without entering a number, the program will not see the Enter and will continue to wait for a number
 - On p. 118, it will skip the fourth prompt, because you hitting enter to move to the hometown prompt after `nextInt` causes the program to think you're skipping that part, or that Enter (blank) is what you intended
 - So...for now, it's best to **enter everything as a String, which reads the entire line, then convert to other types as needed**

Lesson 7: Controlling Program Flow

- If-Else Statements **Listing 7.3**
- Exclusive-Or Operator (^) / XOR checks to see whether the operand on either side is true but also checks to make sure both are not true

Table 7.2

Table 7.3

- Switch Statements
 - The difference from if-else is that if-else statements must test booleans. Switch can evaluate byte, short, char, int, and enumerated types, strings, and wrapper types for byte, short char, and int
 - The switch statement simply checks if the evaluated expression matches any of the cases

- The break statement stops the entire switch and executes the first statement of code after the switch block. This is known as *falling through*
- It is always good practice to always include a default case even if you don't believe it will ever be used. It's a good safety check for your code.
- You can run the same code for several values in a switch statement, or stack up the case statements one after the other

```
switch (day) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        dayType = "Weekday"
        break;
}
```

- Switch logic can be achieved with if and else-if statements. Using it is really just a matter of preference.
- The key difference is that for a switch statement you need to have a list of acceptable values that are being compared to the same thing
- All switch statements can be converted to if-else logic; however, not all if-else expressions can be converted to a switch
- Comparing for equal strings
 - .equals() method
 - Checks to see if the value of a variable is the same as what is in parenthesis
 - Evaluates to true or false
 - For example:

```
dayName.equals("Saturday");
```

- compareTo()
 - Returns an integer.
 - Equal strings returns 0
 - If the string passed to compareTo() is greater than the original string, then a value greater than 0 is returned.
 - If the string passed to compareTo() is less than the original string, then a value less than 0 is returned.
- Strings are sorted based on their characters. Numbers come first, then uppercase letters, then lowercase letters
 - This means that "a" is less than "b"
 - 1 is less than z
 - Zzz would come before (is less than) aaa because Z comes before the lowercase

letter a

Lesson 8: Looping Program Flow

- Curly brackets are not required if the code to be executed is a single statement. The following are both acceptable:

```
while (counter <= 5)
    System.out.println("Counting: " + counter++);
while (counter <= 5) System.out.println("Counting again: " + counter++);
```

- The break; statement send control to the next line following the current loop
- do-while loops
 - Almost identical to a for loop, but their code blocks are executed first and their conditions checked to see whether they should do it again only after it has been run once
 - Syntax
 - Where it shines: asking users for input

```
do
{
    // code to be repeated
} while (condition);
```

Lesson 9: Understanding Flowcharts and Algorithms

Flowchart Basics

- There are three things a program does:
 - Sequence: execute a series of statements in order
 - Branch: Follow a specific path based on a defined condition
 - Loop: Repeat a series of instructions until a condition is met
- Flow of control is defined as the path a program takes from one instruction to the next.
- An algorithm is a well-defined set of instructions that a computer follows to reach a desired output
- A sequence is a control structure that dictates how an algorithm moves sequentially from one statement to the next
- Branches
 - In a branch situation, the program reaches a point where there are two or more possible next steps, and it must evaluate a condition to determine which path to take.
- Loops can also be part of a flow of control. Be careful and avoid infinite loops
- Flowchart symbols
 - **Oval:** Indicates the start and end points of an algorithm

- **Rectangle:** indicates an instruction that the computer must complete
 - **Parallelogram:** indicates input and output
 - **Diamond:** indicates a condition
- You can write pseudocode in the form of an outline.
 - It can be used to supplement or replace flowcharts early in the development process
 - It is simply the process of writing out a program's steps using a more natural human language that anyone can understand
 - In the book's example, each indentation represents a code block, and each of those blocks will be a shape in the flowchart
- Always start any program you write by sketching a flowchart and/or writing out the steps in pseudocode, before you open your IDE. If you can't solve it on a whiteboard, you can't solve it in code. "Measure twice, cut once."

Exercise: A Guessing Game

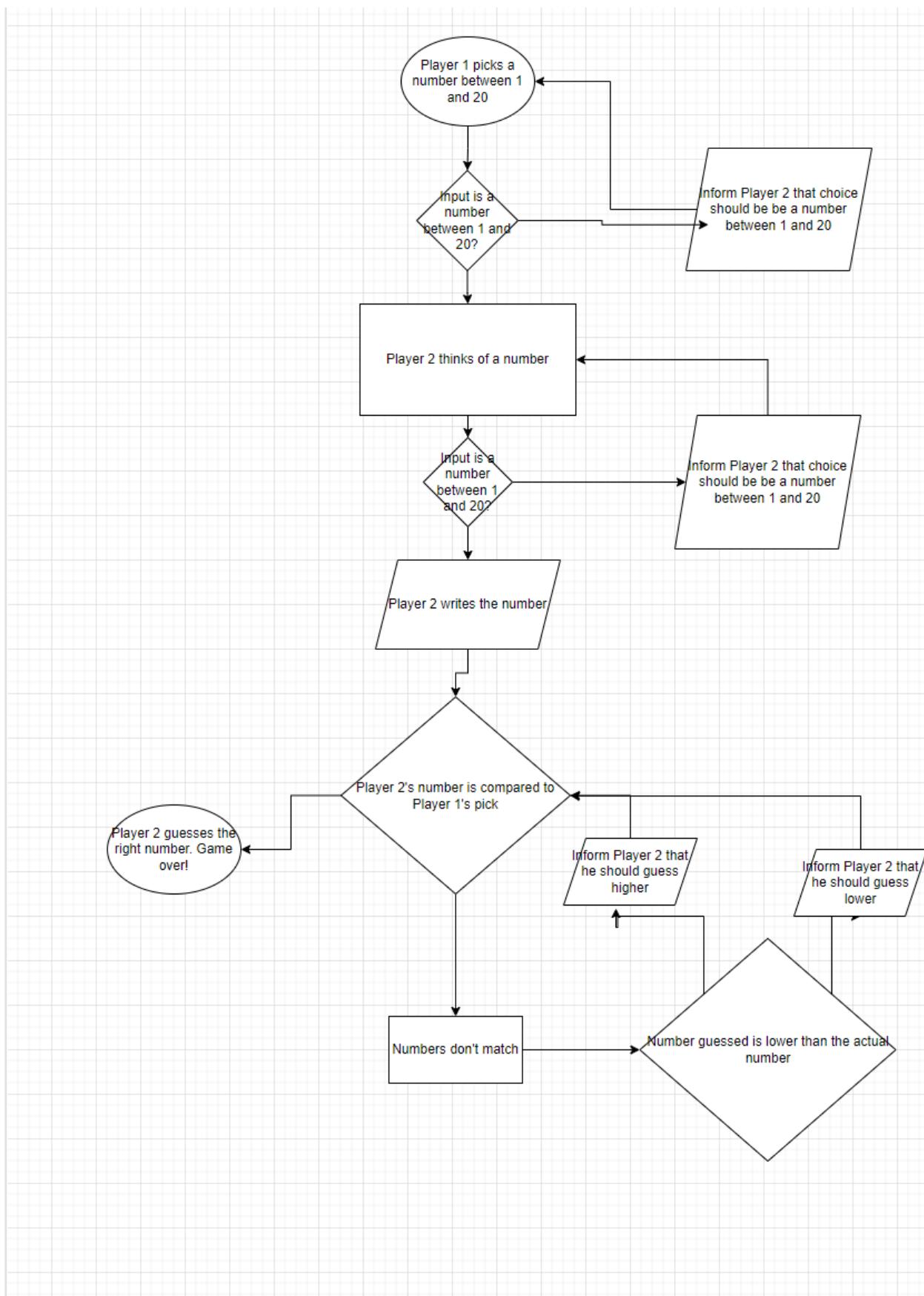
Pseudocode

1. Player 1 thinks of a number between 1 and 20
2. Player 2 writes the number in the prompt
 - a. If Player 2 guesses the correct number, the game ends
 - b. If not, Player 2 must guess again

From the book - things to consider:

What pieces of data do we need to track?

- What player 1 picks
- What player 2 guesses
- That the number is between 1 and 20 only
- How do we know if the number is higher or lower than the actual number chosen?
- Messages need to be displayed



A Possible Solution - how the authors thought it through

- They thought about the data we need to keep track of
 - A variable for the answer
 - Variable for user input (guess)
 - A scanner to read input from the console
- Need an input/output process to prompt Player 1 to enter the number onto the keyboard
= parallelogram
- Player 2 can make guesses. The # depends on how lucky he is, so a for loop wouldn't work. A while loop would be better
- After receiving inputs, was the user right or wrong? Was it in the right range?

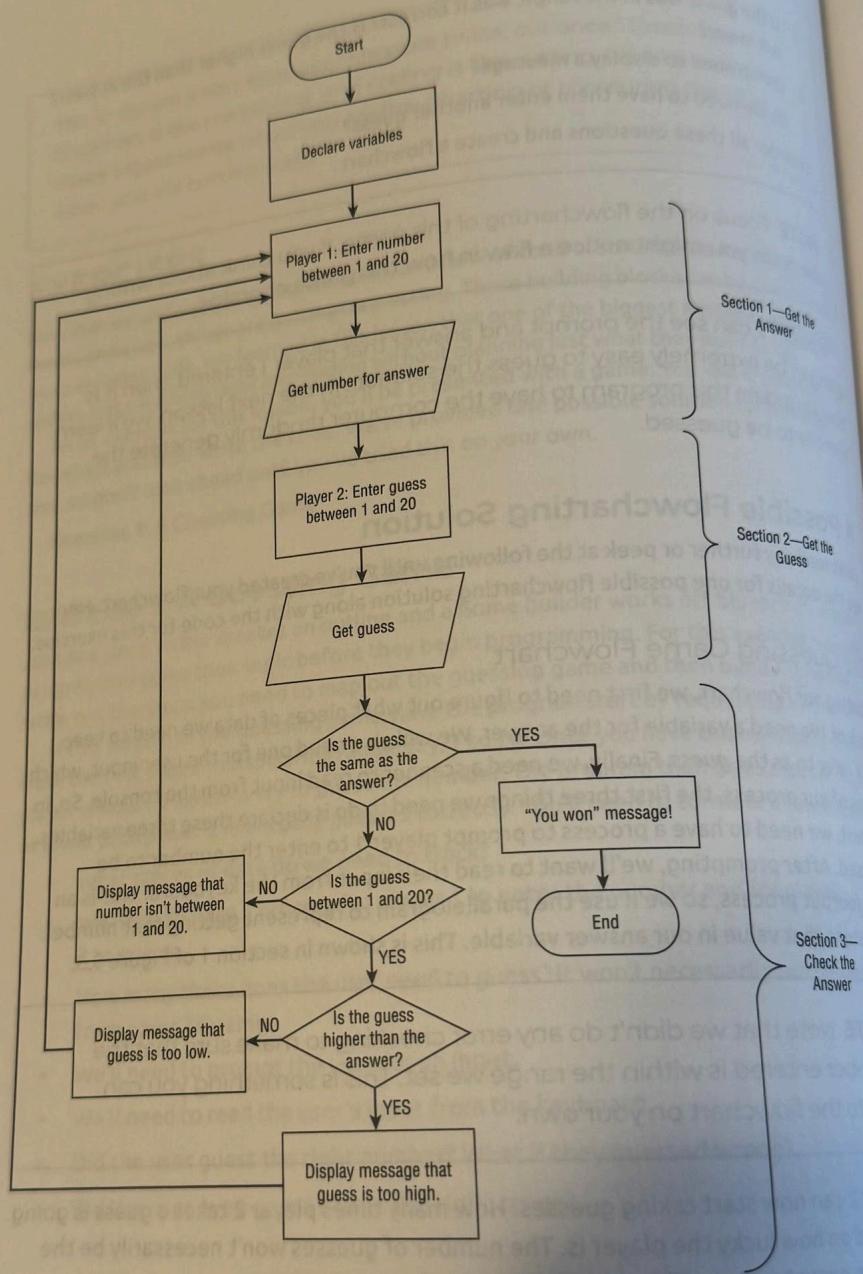


Figure 9.5 The guessing game flowchart

Translating into code - my version

```
int number = 0;
int guess = 0;
Scanner playerOne = new Scanner(System.in);
Scanner playerTwo = new Scanner(System.in);

if (number < 1 || number > 20) {
    System.out.println("Please select a number between 1 and 20);
}
else {
    number = playerOne;
    while (guess != number) {
        System.out.println("Player 2, select a number");
        playerTwo
        I was originally going to do the if else - but DRY
        if playerTwo =
    }
}
```

Note: the *continue* keyword

The *continue* keyword is used to end the current iteration in a *for* or *while* loop and continues to the next iteration.

Chapter 10: Adding Randomness to Your Programs

- We include the Java Random utility package thusly: import java.util.Random;
- We create random numbers through the use of one of two constructors
 - Random fixedSeed = new Random(112358);
 - Uses a predefined seed value, which means the Random object will produce the same set of random numbers each time, and would change if we picked a different integer
 - Not ideal since it's not true randomness
 - Random timedSeed = new Random();
 - Picks a seed based on the date and time it's run
 - Generates a completely different sequence of random numbers
- It is good practice to create a single Random object in your application and use that repeatedly. Multiple objects ruin pseudorandomness
- Generating Random Integers
 - Creates a random int from 0 to a max number exclusive, which means that the number you specify will be excluded from all possible outcomes. For example, if you want to generate integers from 0 to 9, specify an upper range of 10

```
Random rng = new Random();
// generate 10 numbers, each between 0 and 9 inclusive
for (int i=0; i<=10;i++) {
    int randomNumber = rng.nextInt(10);
    System.out.println(randomNumber);
}
```

```
int randomNumber = rng.nextInt(10) + 1
```

Produces a range from 1 to 10.

```
int randomNumber = rng.nextInt(11)
```

Produces a range from 0 to 10

- Generating Random Doubles
 - Similar to integers:

```
Random rng = new Random();

double randomValue = rng.nextDouble();
boolean coinIsHeads = randomValue < 0.5;
```

- The process is different if we want to specify a range outside of 0.0 to 1.0 → we have to do it ourselves
 - Adjust the range to fit the range of our desired outputs (max - min)
 - Shift the now correctly sized range to the right place

```
Random rng = new Random();

double min = 7.5;
double max = 13.2;

double range = max - min;

double zeroToOne = rng.nextDouble();

double stepOne = zeroToOne * range; // adjusting the range
double stepTwo = stepOne + min;      // shifting the range
```

```
double desiredOutput = rng.nextDouble() * (13.2 - 7.5) + 7.5
```

- Another way to produce randomness is Math.random()

Lesson 11: Debugging

- Breakpoints are places in the program where we are telling the debugger to execution so we can make sure things look the way we believe they should look. Click the leftmost margin of your IDE to set a breakpoint
- In NetBeans, a green highlighted line means the code was executed until the breakpoint. From this point, two things we can do are **Step Over** or **Continue**
- Step Over
 - Means “execute this line of code, go to the next line of code, and then stop.”
 - Execution will pause again on the next executable statement
 - Shortcut is F8
- Continue means we just continue with program execution. Shortcut is F5.
- While debugging, if nothing seems to be happening, you might need to provide input
- We can change variable values in the Output window
- Again, the main point of what we did is look at the individual execution of each line of code and see what each line was doing.

Lesson 12: Adding Methods

- One of the most important concepts in coding is DRY: Don’t Repeat Yourself
- Methods are an organizational tool that allow us to write some code, give it a name, and then use that code over and over again
 - Allow use to reuse code we’ve written
 - Allow us to break down large, complex tasks into simpler steps
- Defining Methods
 - A method is defined by a method declaration, made up of the following:
<access modifier> static <return type> <method name> (<parameter list>
<exception list> {
 <method body>
})
 - Terms
 - Access Modifier: public is an example of this
 - The static keyword
 - Return type: Indicates the data type of the value returned by the method. Void is used if the method does not return a value
 - Method name: This is the name for the method. The rules for identifiers apply here; additional conventions will be covered at a later time
 - Parameter list: Comma-delimited list of input parameters
 - Each parameter consists of a data type followed by an identifier
 - Must be enclosed in parentheses
 - No parameters? Use empty parentheses

- Exception list: to be discussed later
- Method body: the code block for the method

Figure 12.1

- Method Signature
 - Consists of a method name and a parameter list – memorize it!!
- Method Naming Conventions
 - Should begin with a lowercase letter
 - Should be a verb if a single word name
 - Should be in camel case
 - Follows the naming conventions for other identifiers
- Return Values (check before the method declaration to check the type it's returning)
 - No return value (void)

```
public static void printHiYa() {
    System.out.println("HiYa");
}
```

- Return value

```
public static double calculatePi() {
    return 22/7;
}
```

- Input Parameters
 - Placeholders for values (declarations for the variables) that will be *passed in* as data for the method to use
 - Example

```
public static int add(int a, int b) {
    return a + b;
}
```

- The number of parameters and their data types *must* match the type and number defined by the method; otherwise, the code for the method call won't compile.
- Method Forms
 - A method can take 4 forms:
 - No return value, no parameters
 - Return value, no parameters
 - No return value, one or more parameters
 - Return value, one or more parameters
 - `System.out.println()` is not a return value; it merely prints to the console
- Refactoring Repeatable Code into Methods

- Refactoring is simply the process of reorganizing and/or clearing up your code without adding, subtracting, or changing functionality so it is more readable and maintainable
 - Step 1: Divide out the similarities and differences of the repeated code.
Will represent the main body of the new method
 - Step 2: Decide on a name
 - Step 3: Decide whether the method needs input parameters. If so, how many and of what type?
 - Step 4: Decide whether the method needs to return a value. If so, what type?
 - Step 5: Write the code for the body of the method
- We can have as many methods as we want in a class. Any additional methods we are creating get declared outside the main method, but inside the code
- Scope
 - See the WindowMasterRefactored code - why do we have two Scanners?
 - Java allows us to define variables inside a block of code, which are **local** to that block of code.
 - Variables declared in outer blocks can be accessed by code inside nested code blocks
 - The reverse is not true: code in outer blocks cannot “see into” nested blocks
 - See **ALookAtScope**
- More About Debugging
 - Two other things we can do with debugging
 - How we can set a breakpoint inside a method and have it go to that breakpoint every time even though we’re executing inside the main method
 - How we can **step into** something
 - Step Into
 - Click the Step Into button or press F7
 - This option takes us into the code that is highlighted instead of taking us to the line that follows it
 - Only variables show in scope will appear in the *Variables* window

Lesson 13: Organizing with Arrays

- An array is a set of uniform data elements that can be accessed using indexes
 - An element is a single item in an array
 - Uniform means that all elements in a given array must have the same data type
 - The index of an element refers to its position in the array
 - Think of arrays like a row of lockers
- Array Dimensions
 - They can have any number of dimensions.
 - A 1D array is a vector
 - A 2D array is a table or multidimensional rectangular array

- A 3D array is a cube
- We use the term *rank* to describe the number of dimensions in an array
- Each dimension in an array has a length, or the number of positions in that direction
 - **Figure 13.2**
 - The total length of an array is the total number of elements in all directions. (Figure 13.2 has a length of 12.)
- Another type of array is a multidimensional jagged array, in which each subarray is a single independent array and can have different lengths (**Figure 13.6**)
- In Java, array size is fixed. Lengths cannot be changed after they are defined.
- Element vs. Index
 - Elements of the array are where data is stored
 - Indexes of the array are the integers that define where the data is stored
 - Think of it like this: a house's address (index) allows us to go to the right part of a street (array) and view the house (element) at that location.
- Declaring Arrays
 - Single-Dimension Arrays
 -

```
int[] arrayOfInts;
```

- Rectangular Arrays (each set of brackets determines the array's dimension. Can be considered an array of arrays)

```
int[][] my2DArray = new int [4][4]
float [[[ ] ] ] my3DArray = new float [5] [5] [5];
```

- Jagged Arrays (can be considered an array of arrays. The below is set to two rows but the number of columns has not been stated.)

```
int [][] myArray = new int [2] []
```

- Initializing Arrays
 - If we initialize an empty array, we specify the number of elements we want to allocate space for (like blocks of memory) without providing the data to be stored immediately.
 - Or, we can provide data right away, and the array size will be inferred and set to exactly the size required for that data
 - Because array sizes are set when initialized, it's best to plan out an array on paper before coding it
 - Initializing a Single-Dimensional Array
 - Requires setting up the memory area for values to be stored. We can do this with the *new* keyword

```
int [] numbers = new int [4];
```

- Explanation: We declared a single-dimensional array called numbers. The new keyword is used to initialize the array. It is being created as an array of 4 integers. No values are indicated, so for now each value in the array is set to 0
- When we declare an array, we use curly braces to enclose the block of default values

```
int [] numbers = { 20, 33, 4444, 123};
```

- Initializing a Rectangular Array
 - The following creates an array with two rows and three columns:

```
int rows = 2;
int columns = 3;
int [][] table = new int [rows] [columns];
```

OR

```
int [] [] table = new int [2] [3];
```

- **Figure 13.8**
- The designation of the first dimension as a row and the second as a column is arbitrary. They are just memory locations. But consistent conventions are critical to maintain!
- You can also auto-initialize rectangular arrays

```
int [] [] = autoInitTable = { { 5, 3, 1 } }, { 2, 4, 6 } };
```

- Initializing a Jagged Array
 - Includes multiple arrays of varying lengths
 - Do not have a regular shape, so initialization and declaration is trickier:

```
int [][] jagged = new int[3] [];
```

To initialize these rows, you then need to set each of the rows to an array

```
jagged[0] = new int[] {1,2};
```

```
jagged[1] = new int[] {3,4,5,6};  
jagged[2] = new int[] {7,8,9};
```

- If you were to have more than two dimensions in a jagged array, it is recommended that only the last dimension be jagged
- Accessing Elements in a Single-Dimensional Array - this I can do
- Accessing Elements in a Multidimensional Array
 - **Figure 13.2**

int gottenNumber = autoTable[0][1];
○ To assign the item in the second row and third column to a variable, we could do:

```
int anotherNumber = autoTable[1][2];
```

- **Rows first, then columns**
- Accessing Elements in a Jagged Array
 - **Figure 13.13**
 - To get the second element from the third array:

int eight = jagged[2][1];

○ To replace the value in the second position of the first array:

jagged[0][1];
- Iterating Through Arrays
 - We either loop through the valid indexes and use them to get the element, or we iterate through the elements directly (typically if we don't know the index)
- Dealing with Errors
 - Index out-of-range exceptions are the most common exceptions you encounter when dealing with arrays (we try to access an element of an array that does not exist)
 - If you get this error when iterating through an array, it usually means that your loop's conditional expression is incorrect

Lesson 14: Object-Oriented Concepts

Object-oriented languages represent concepts in a solution and problem space as objects; in other words, they model objects (like a car) using properties and behaviors. In OOP:

- Everything is an object
- A program is a collection of objects that tell each other what to do by sending messages (in Java, method calls)
- Each object can be made up of or composed of other objects (in Java, this is called

composition)

- Every object has a type
- All objects of a particular type can receive the same messages (they all have the same methods)

IN other words, an object has state (has certain values at certain times), behavior (an object has a capacity to do something), and identity (able to be distinguished from other objects even when they are the same type i.e. the shade of a tree in a front yard and a backyard)

Types

- Everything in Java has a type, or a classification that defines the structure and range of values for the type and the associated operations allowed on those values
 - Java has intrinsic types (int, double, boolean, etc.)
 - We can also define our own types, i.e. every time we define a new Java class

Public Interface/Private Implementation

- The definition: every class should have a public interface that defines how the outside world can interact with it. Behind this public contact should be a private implementation which allows us to separate “what” and object does from “how” it does it.
- Calling code (other than the parts of the program that use the object) should not be concerned with how an object fulfills a contract and should in no way ever rely on the specifics of the implementation when using the object
- The analogy: ordering fast food at a drive thru
 - Public: the process (look at the menu, order into the speaker, pay, get food)
 - Private: How the order is processed, the OS used, the number of workers or stoves

Encapsulation and Data Hiding

- Faciliates the notion of public/private
- Well-designed classes prevent direct access to their properties by calling code and force access from other parts of the program using getter and setter methods.
 - This prevents the calling code from being aware of the internal details of the object
 - Allows the internal representation of the properties to change without the knowledge of the calling code (called loose coupling)

Single Responsibility Principle and Cohesion

- A well-designed class has a well-defined area of responsibility
 - The class should fully contain all aspects of its area of responsibility
 - The public interface must be defined so that its function is crystal clear, even though how it is implemented is hidden
- Drive-thru example
 - If the interface is cohesive, it must allow us to do everything involved in ordering, paying for, and picking up our meal,

- I should not have to get out of the car to transfer funds (not cohesive)
- I shouldn't be able to renew my driver's license at the payment window (not fulfilling single-responsibility principle)

Delegation

- Delegation complements encapsulation
- If one or more tasks within the class's area of responsibility requires a subtask that is outside the class's main area of responsibility, the class must delegate that task to another class
 - Example of this is System.out for writing to the console or Scanner.in for reading input
 - For example, the drive-thru that doesn't specialize in baking bread for buns "delegates" this responsibility to a baker that bakes the buns and delivers them to the restaurant. The customer doesn't care where the buns are baked, just as long as they taste good!

Lesson 15: Creating Classes and Types

- Reminder: there are two categories of data types in Java: primitive and reference (new classes)
 - Types consist of fields (properties) and behaviors (methods)
 - Fields are sometimes referred to as members
- See Listing 15.1 in NetBeans
- Classes vs. Objects
 - A class is a definition, like the blueprint of a house
 - A blueprint is a detailed model of a building
 - It shows you how to build the house, but you can't live in a blueprint. You have to follow the blueprint to build the house before you move in
 - Similarly, you must instantiate an object, based on the definition contained in the class, before you can use it
 - A class is an idea, an object is the instantiation of that idea. You can pet a Maltese Shitzu named Hershey, but you can't pet an idea of a Maltese Shitzu
- Properties, Accessors, and Mutators
 - A common technique used to hide data in Java is to use accessors (getters) and mutators (setters). They are methods that get and set the values of the properties (or fields) on an object
 - The process of using getters and setters for a field is to declare the field as private and then create methods that start with get and set
 - What's the point? Why not let clients access and change the values of our class's properties directly?
 - It is desirable for code that uses an object to have no idea how the properties are stored or calculated
 - The code should just know what each getter and setter does
 - In this way, we create self-governing objects. Any changes made to the

state of the object (aka changes to properties) are made only by the object's methods

- Example
 - A Student class might have an ID property
 - This is typically generated by the school and can't be changed by anyone else
 - In this case, it's best to have a private implementation of ID and then implemented a getter to return the ID (no need for a setter since we don't want any entity to change it)
- The dot operator (.)
 - Used to access visible properties or methods of an object
 - Used for static and nonstatic properties and methods
 - Left side is class (static) or variable (nonstatic) field or method
 - Right side is the method or field we want to access
- this Keyword
 - Used to refer to the **fucking instance of the fucking class** in which the **fucking code is currently fucking executing**
 - Used **in conjunction with** the dot operator to access properties and methods of the containing class
 - Static methods and variables should **not** be referred to with the this keyword
 - The this keyword cannot exist outside constructors and instance methods of a class
- Methods/Behaviors
 - Classes can have properties, and they can have **behaviors**, which are implemented as methods
 - Methods are always contained inside a class definition. They cannot stand on their own
 - The methods in a well-designed class will match the purpose of the class (i.e. a Dog class should not have a method called meow())
- Constructors
 - A special method called when you create an instance of class
 - Are usually used to initialize the properties of a newly instantiated/created object
 - Special Rules
 - Must have the same name as the class that it is a part of (the constructor for a class called Dog would be Dog())
 - Never have a return type, not even void
 - Can have parameters but don't have to
 - There can be more than one constructor in a given class
 - You don't have to create a constructor for your class as long as the superclass has no-arg constructors
 - If you don't create one, the compiler will supply one called the default constructor, which has no parameters and appears to have an empty method body
 - Such a constructor is the default constructor whether you write it or

the compiler provides it

- See Listing 15.2
 - If we use the default constructor, with no values set on the fields, the fields get initialized to their defaults (null, 0, false, etc.)
 - We would have to invoke setter methods to set the values to what we want
 - Alternatively, if we use the custom constructor, we can instantiate the object with the values we choose
- Recap
 - If we have variables of local scope, we must initialize them before they can be used and we must do that ourselves
 - But when we instantiate a class, any of the fields in the class that are not initialized in the constructor will get their default values (i.e. 0 for primitive types)
- static Keyword
 - If a field or method is marked as static, it means that it is associated with the class and not with any particular instantiation of the class
 - In other words, there is only ever one copy of a field or method that is marked static, whether 0 or 97 instances of a class are created
 - For example, if the Dog() class was static, all Dogs objects would share the same name
 - Static fields and methods can be accessed without creating an instance of the class
 - Nonstatic properties and methods are associated with a particular instance of the class, which means they are accessible only through that particular instance of that object. This also means that nonstatic fields and methods do not exist until one or more instances of the class have been created
 - When is it appropriate to mark a method as static?
 - Main method
 - Again, the main method is the entry point of our program, or where it kicks off
 - Since this is the first thing that is going to run in our program before any objects are instantiated, it must be static
 - Constant values
 - If there will ever only be one copy of that constant in our program because the value can't be changed, it makes sense to mark it static
 - Example: PI
 - Utility methods
 - Examples of utility methods are those that perform math operations
 - Methods marked static must not attempt to change the state of the class with which they are associated (i.e. math operations take inputs, operate on the inputs, then produce outputs)

Listing 15.3: Using Static Constants and Methods

```
public class SimpleMath {  
  
    public static final double PI = 3.14;  
    public static int add(int a, int b) {  
        return a + b;  
    }  
  
    public static int subtract(int a, int b) {  
        return a - b;  
    }  
}
```

- As shown above, we don't need to have to first instantiate the object and can use it directly from the class, instead of Scanner, where an object first has to be created before it can be used
 - Factory pattern
- A good example of a library of static methods together in classes is Math().

Lesson 16: Managing Storage and Memory

Programs and Memory

- When your program is running, its data must be stored in memory
- The JVM uses two main areas to store the data associated with our programs: the stack and the heap
- The memory that each method gets called is a stack frame
- Any new object created by our program will get some memory on the heap

The Stack

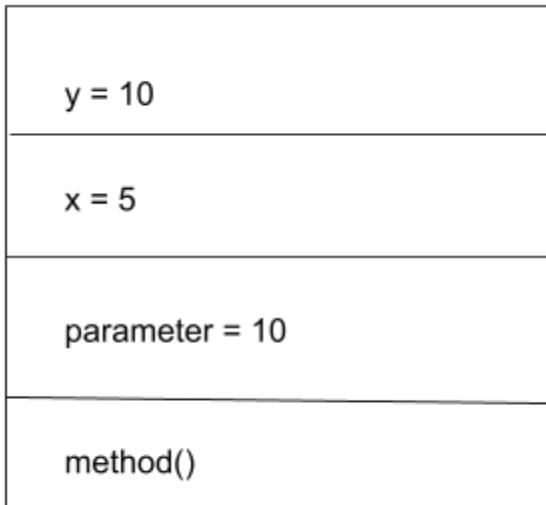
- A LIFO data/memory structure managed by the JVM (think of plates at a restaurant buffet)
- New items are added to the top (pushing the item onto the stack)
- Items are also removed from the top of the stack (popping the item off the stack)
- Instead of restaurant plates, the JVM adds a stack frame for each method that gets executed in our program. The stack frame contains the following information
 - Values of all primitive type variables declared in the method
 - The heap location of any nonprimitive types declared in the method
 - Values of the parameters passed into the method
 - JVM bookkeeping information about the method
- An Example

```
public void method (int parameter) {  
    int x = 5;  
    int y = parameter;  
}  
  
// Let's call the method with a parameter of 10  
method(10);
```

- The above method takes an int parameter and creates two more integers
- The calling code invokes the method and passes a value of 10 to the parameter variable
- Because these are primitive types, the data and variable name are stored on the stack
- The method() entry on the stack represents the bookkeeping information the JVM needs to keep track of the execution environment

Method

Stack



- When `method()` is done executing, the memory on the stack will be popped off it, which will cause the remaining stack items to then be deleted from memory

- If method() is called again later in the program, the JVM will build a new stack frame and push it onto the stack
 - Primitive types will have their data value physically copied into the new memory space
 - Changing parameter, however, won't affect the value of y because y and parameter are contained in two separate memory spaces in the stack frame

The Heap

- An area where chunks of memory are allocated to store reference types
- Things can be added and removed in any order
- Requires two segments of memory
 - The heap contains the actual data (some type of object)
 - The stack contains a pointer to the actual data. The value of the pointer is the address location in memory where the heap's data can be found
- It is more efficient to store data in a heap when dealing with large class objects, for instance, where only the reference is copied once from the heap to the stack. Otherwise, if we copied class objects like we did with type int on the stack, the computer would quickly run out of memory

Listing 16.1

```
public class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

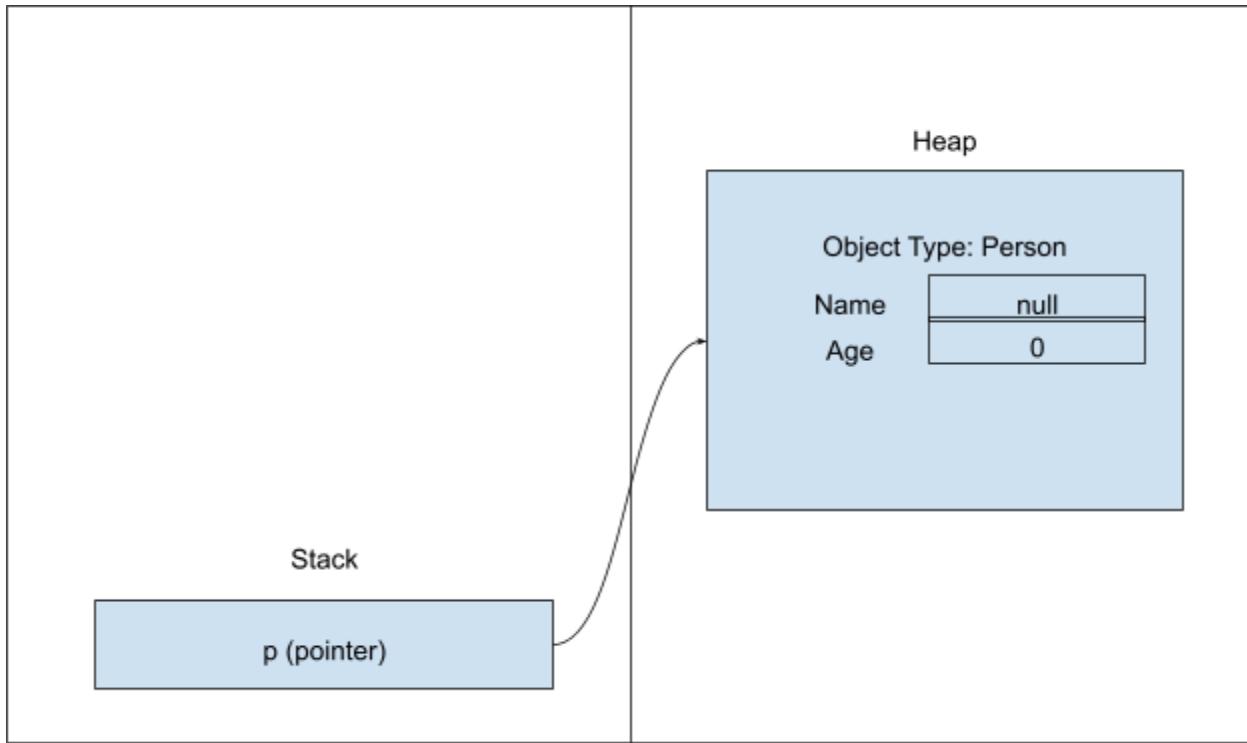
    public void setName (String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge (int age) {
        this.age = age
    }

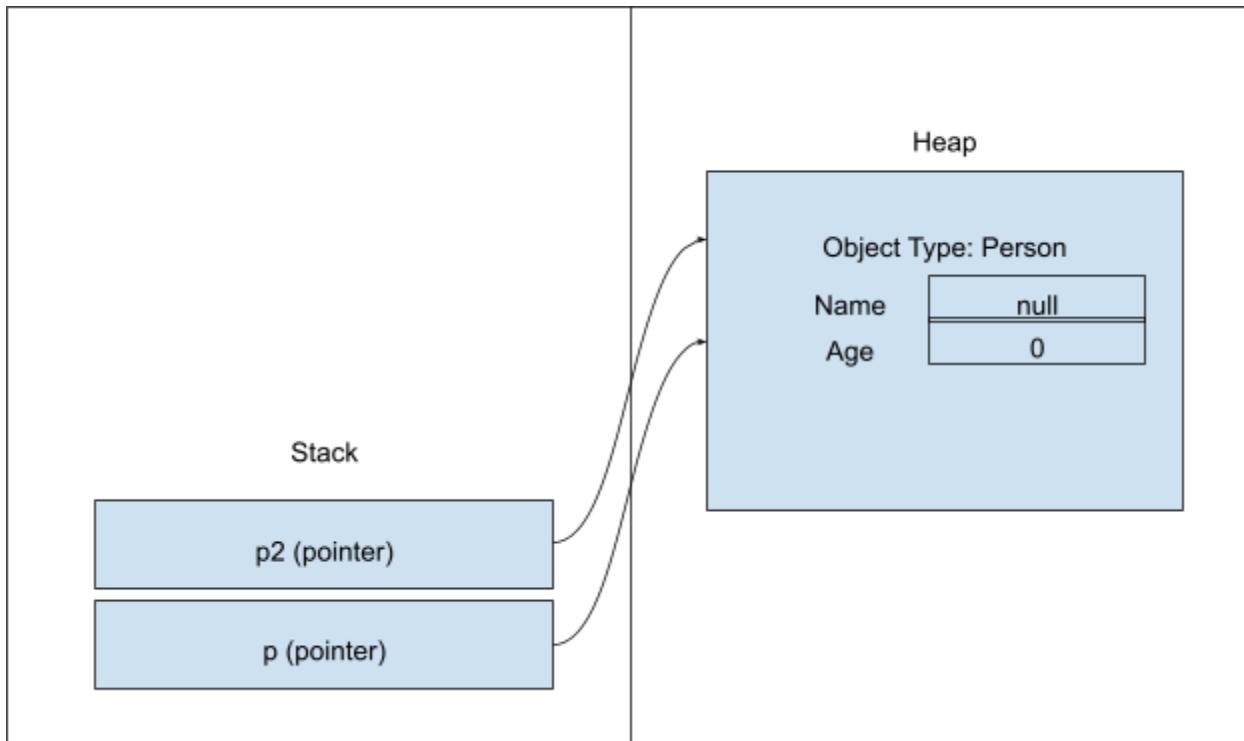
    Person p = new Person();
```

When the new keyword and one of the constructors of a class are used together, JVM creates a new object of that particular type on the heap and hands us back a reference to this object. The below shows the stack and heap with a constructed Person



If we added a second person: `Person p2 = p` → the JVM would only copy the pointer and not create a second object

- In other words, both **p** and **p2** point to the same object on the heap
- If we assign a name to **p** and an age to **p2**, both would see the data updates



To Recap

- Primitive data types are stored on the stack. When data is passed by value, a copy of the data is made, such that changes to the value do not affect the other copies
- User-defined types are stored on the heap with only a reference stored on the stack
 - When a reference is passed to a method, a copy of the reference is made, meaning the object on the heap now has one more reference pointed to it
 - Changes to the data made through any reference to that object are seen by all other references to that object
- Java is a pass-by-value language, meaning that a copy is made of any value passed into a method
 - Primitive type's actual value is copied into a new memory location on the stack
 - User-defined type's reference value is made and stored in a new memory location on the stack

Garbage Collection

- Java is a managed language, which means the programmer is not responsible for allocating and releasing memory manually. (This is not the case for a language like C++; when not allocated and deallocated properly, this can cause memory leaks and application crashes.)
- All memory allocated on the stack is automatically released when a method completes, and its associated stack frame is destroyed
- Garbage Collection
 - Memory used by objects on the heap are eligible for this when there are no more

- references to the object
- We can explicitly take a reference away from an object on the heap by setting it to null
- Garbage collection refers to the process where reference variables on the stack go away when their respective stack frames (when their enclosing method returns), which also removes the references from the object to which they point.
- The garbage collector is responsible for this, and it periodically sifts through the objects on the heap and check to see whether they are still being referenced on the stack. If not, the objects may be marked for cleanup and removed.
- Garbage Collection is one of the key features of managed languages and is a reason why you would choose a language like Java or C# over C++

Reference and Null Values

- Again, primitive data types are considered value types. When created, they are given default values. (i.e. numerical values default to zero)
- User-defined types, aka reference types, have default values set to null
- **Null** is a special value in Java that indicates we have a user-defined type that is not associated with any object on the heap. In other words, we have an object, but it's not pointing to anything
- NullPointerException
 - An error in the code
 - If we have a reference to an object, it is an error to try to operate on it if it's not actually pointing to something on the heap
 - It's like having a remote control without a television. You can use it, but what good will it do you?
 - The fix? *Make sure that we have instantiated our reference and that we are pointing to a real object on the heap.*

Lesson 17: Exploring Interfaces, Composition, and Inheritance

The Benefits of These Tools

- Allow us to control how other code interacts with our objects
- Allow us to define relationships between the objects in our programs
- Help us keep our implementations private
- Help us integrate code from several teams or vendors
- Allow us to develop different pieces of large applications in a distributed manner
- Help us to follow DRY

Interfaces

- In this chapter, an interface refers to a specific feature of the Java language
- Think of an interface as a contract. We can create Java classes that promise to implement all the capabilities described in the interface
- They are good at expressing a loosely coupled, contract-based relationship. These types of relationships are especially good at enabling interoperability, and they give us a

mechanism to protect our private implementation. This is especially helpful when integrating code from different teams or vendors

- An example where an interface is used to enable interoperability and integration: making cars self-driving
 - We could integrate an external vendor's software for each vehicle in our lineup, one-by-one...
 - Or, we can create a single navigation interface that works on all cars
 - Only needs to be integrated one time
 - Each underlying implementation would be different and could be changed at any time without affecting the overall self-driving system
 - But the interface describing how to control and navigate our vehicles would be the same
- A class may implement as many interfaces as it needs

Composition and Inheritance - both allow us to express relationships between objects and to reuse code. You must first determine the type of relationship you want to express, taking account the different approaches of each, before you can decide which one to use.

Composition

- The mechanism that allows objects to be made up of other objects
- Expresses a **has-a** relationship. It allows us to reuse code by creating fields in our objects that are other objects. We then delegate to the contained object when we want to take advantage of the capabilities of that object.
- Example - fast food restaurant
 - The architect doesn't design the stove; he makes a space for it
 - The contractor doesn't build the stove from scratch; she orders it
 - The restaurant **has** a stove, but it no one would say the restaurant **is** a stove
- Composition is great at these kinds of relationships where you need the capabilities of another object, but that's as far as the relationship goes.

Inheritance

- A core feature of OO languages. In fact, without it, a language can be object-based but **cannot be object-oriented**
- Inheritance allows us to express an **is-a** relationship between two types. These types of relationships express a hierarchical relationship or taxonomy.
- Example - motorcycles
 - All motorcycles have two wheels, a motor, handlebars, etc.
 - A dirt bike has all the characteristics of a motorcycle, plus off-road tires and high-performance suspension
 - A street bike has all the characteristics of a motorcycle, plus a headlight, license plate, etc.
 - We want the latter two objects to inherit all the features of a motorcycle but add in the special parts.
- Both a dirt bike and street bike have an is-a relationship with a motorcycle. In OOP, the

motorcycle is the parent of both dirt and street bikes. Dirt and street bikes **extend** motorcycle

Using Interfaces, Composition, and Inheritance - Rules of Thumb

- We almost exclusively use interfaces and composition in software we write for this course and web app development in general
- The code we write will use classes that have inheritance relationships all the time. Every class you create will have an implicit inheritance relationship with a class called **Object**.
- Inheritance is often used in the frameworks and libraries we will use to build our applications.

Lesson 18: Diving into Interfaces

What is an Interface?

- Again, an interface is a reference type, like classes and enums. **It defines a set of methods that provide standard behaviors that any type that implements the interface must provide.**
- However, interfaces are never instantiated → you can't do `new SomeInterface()`

Declaring an Interface

- See Listing 18.1
- Compared to a class declaration, the key differences when declaring interfaces are as follows:
 - The *interface* keyword is used instead of the *class* keyword
 - An interface can have abstract methods, default methods, and static methods
 - Abstract methods are methods that are declared without an implementation
 - A semicolon is placed after each method definition
 - In the previous example, none of the defined methods has implementations because they are abstract methods. (We can use the *abstract* keyword, but it's implicit.)
 - Default, static, and private methods can have implementations in an interface
 - All abstract, default, and static methods in an interface are implicitly public
- Can include method declarations. All types of methods can be declared in an interface - with or without return types, with or without parameters, and in any combination

Implementing an Interface

- To have an interface doing anything useful, we must create a class that promises to fulfill the contract defined by the interface. In other words, the class **implements** the interface
- You cannot instantiate a standard interface on its own; it requires an implementing class
 - The class must declare that it implements a given interface
 - The class must provide implementations for each of the methods defined in the interface
- See Listing 18.2
 - The MidsizeSedan class has methods that align with those in the interface

- The `@Override` symbol is an **annotation**, a form of metadata that helps describe the program but isn't part of the program itself.
 - The compiler can use annotations to help detect errors
 - Code generation tools can use annotations
 - Some annotations are available to be inspected at runtime

Interface Restrictions

- Interfaces have restrictions because they are meant to be contracts
- Interfaces cannot have member fields, but they may define constants
- None of the methods can have implementations. However, from Java 8 onward, FYI, interfaces are allowed to have default implementations of methods, which can be useful in certain contexts

Implementing Multiple Interfaces

- A class may implement more than one interface
- Such a class must simply list all the interfaces that it implements and then provide implementations of all the methods of each interface
- See Listing 18.3 & 18.4

Extending an Interface

- Useful when we want to define an interface that has the same capabilities as an existing interface, but with one or more methods
- See Listings 18.5 and 18.6

Interfaces and Polymorphism

- Polymorphism means “many-formed” - an object can take more than one form.
- Objects can be treated polymorphically when they implement an interface or when they use inheritance to extend another class.
- Examples
 - MidSizeSedan implements two interfaces: Vehicle and Trackable
 - `MidSizeSedan car = new MidSizeSedan();`
 - New MidSizeSedan object is created on the heap and point to it with MidSizeSedan reference
 - Reference variable (car) as access to all the methods on MidSizeSedan
 - `Vehicle vehicle = new MidSizeSedan();`
 - New MidSizeSedan object also created on the heap
 - This time, we point to it with a Vehicle reference
 - Since we have chosen to treat the MidSizeSedan as Vehicle by referring to the MidSizeSedan object with a Vehicle reference, *we only have access to the methods on the MidSizeSedan that are defined in the Vehicle interface*
 - `Trackable trackable = new MidSizeSedan();`
 - New MidSizeSedan object also created on the heap
 - This time, we point to it with a Trackable reference

- Since we have chosen to treat the MidSizeSedan as a Trackable by referring to the MidSizeSedan object with a Trackable reference, we *only have access to the methods on the MidSizeSedan that are defined in the Trackable interface*
- In each of the previous examples, a new, complete MidSizeSedan object is created on the heap. The difference is that we refer to the newly created object with different reference types, which determine how the newly created object looks to us.

Lesson 19: Diving into Composition

- Composition means that one or more of the fields in an object are other objects rather than primitive types or strings.
- One of the main characteristics of an OO language is that objects can be made up of other objects
- Example: Book without Composition
 - Class keeps track of the minimum data required, but...it's a lot
 - All the information is in separate fields, and there is nothing to tie the author or publisher information

```
public class BadBook {
    private String title;
    private String isbn;
    private String authorName;
    private String authorStreet;
    private String authorCity;
    private String authorState;
    private String authorZip;
    private String publisherName;
    private String publisherStreet;
    private String publisherCity;
    private String publisherState;
    private String publisherCity;
    private String publisherState;
    private String publisherZip;
    private String publisherPhone;
}
```

- Example: Book with Composition
 - We need to keep track of:
 - Book
 - Author
 - Publisher
 - Address (both authors and publishers have an address)
- An Address Class

```
public class Address {  
    private String street;  
    private String city;  
    private String state;  
    private String zip;  
}
```

- An Author Class
 - We are now using composition
 - The Author class has-an address
 - We put the appropriate info in the object and create a field in the Author class of type Address to hold that info - keeps all the address info in a single container

```
public class Author {  
    private String firstName;  
    private String lastName;  
    private Address address;  
}
```

- A Publisher Class

```
public class Publisher {  
    private String name;  
    private Address address;  
    private String phone;  
}
```

- The Book Class
 - We are using composition on two levels
 - The Book class has-a publisher and has-an array of authors
 - The Publisher and Address classes each have-an address
 - We are using an array to hold the Author objects for our book

```
public class Book {  
    private String title;  
    private String isbn;  
    private Author[] authors;  
    private Publisher publisher;  
}
```

*I did Exercise 19.1 - see NetBeans for my solution, which was on the right track, but not ideal.
See ChatGPT's critique:*

Your approach to breaking down the Classroom class using Composition is on the right track, but there's a small issue with how you've structured the Teacher class. Let me explain:

1. Classroom Class:

- You simplified the `Classroom` class by creating arrays for `Teacher` and `Student`, which is a good approach. However, the `Teacher` class in this context should not hold a reference to the `Student[]` array. The `Classroom` is responsible for managing both the teacher(s) and students.

2. Teacher Class:

- You've included a reference to `Student[]` within the `Teacher` class. This can create redundancy and potential confusion, as the `Classroom` class already manages the students.
- Also, a teacher doesn't need to manage students directly; instead, they should manage the class they are assigned to (class name and grade level).

3. Student Class:

- Your `Student` class looks good as it encapsulates the student-related attributes effectively.

Suggested Revision

Here's a refined version of your approach:

```
public class Classroom {  
    private int maximumNumberStudents;  
    private Teacher teacher; // Assuming one teacher per classroom  
    private Student[] students;  
  
    // Constructors, getters, and setters  
}  
  
public class Teacher {  
    private String firstName;  
    private String lastName;  
    private String className;  
    private int classGradeLevel;  
  
    // Constructors, getters, and setters  
}  
  
public class Student {  
    private String firstName;  
    private String lastName;  
    private int grade;
```

```
// Constructors, getters, and setters  
}
```

Key Points:

- The `Classroom` class has a single `Teacher` object and an array of `Student` objects.
- The `Teacher` class now only handles the teacher's details and the class they are assigned to, not the students directly.
- The `Student` class remains the same.

This structure ensures each class has a clear responsibility, which makes the code easier to manage and understand.

40

You said:

Got it. But, don't teachers "have-a" student(s), too?

ChatGPT said:

ChatGPT

You're right that in a real-world scenario, teachers "have" students in the sense that they are responsible for them. However, in object-oriented design, Composition should reflect clear, distinct responsibilities for each class, avoiding redundancy.

Here's why it's often structured the way I suggested:

1. **Separation of Concerns:** The `Classroom` class is responsible for managing the relationship between students and the teacher. By placing the `Student[]` inside `Classroom`, you centralize the responsibility of managing which students belong to which teacher in a single place.
2. **Redundancy Avoidance:** If both `Teacher` and `Classroom` classes have a `Student[]`, you might end up with duplication of data, which can lead to inconsistency. For example, if you change the student list in `Classroom` but forget to update it in `Teacher`, your data becomes inconsistent.
3. **Flexibility:** By keeping `Student[]` in `Classroom`, the structure is more flexible. It allows you to easily handle scenarios like:
 - Multiple teachers sharing the same students.
 - Changing the association between teachers and students without altering the `Teacher` class.

Lesson 20: Diving into Inheritance

- Everything Extends Object
 - The Object class, with its 12 methods, is the parent of every class in Java
 - It is at the root of the class hierarchy; the relationship is automatically defined when you create a class
- Terminology
 - Parent class
 - Also known as the superclass or base class
 - When classes extend a parent class, they inherit the properties and behaviors of the parent class
 - Child class
 - Also known as the subclass, extended class, or derived class
 - The class inherits the properties and behaviors of the base class when it extends the base class
 - This class can add properties and behaviors to those of the base class
 - This class can override (i.e. provide its own implementation of) properties and behaviors of the base class
 - We say this class specializes the base class
- Access Control
 - The protected keyword
 - If a property or method is marked protected, it means that property or method can be seen by the base class, all derived classes, and other classes in the package
 - However, it cannot be seen by any other class
 - Package-Private Access
 - The access level that applies when you have no access modifier for one of the members of your class
 - Grants access only within the class and other classes in the package, both of which can be okay, but prevents access from subclasses
 - Use Cases
 - Unit testing
 - Where several classes in a package work together but you want to restrict access to some members to only classes in that package, or when a class is final and should not be extended

Modifier	Class Access?	Package Access?	Subclass Access?	World Access?
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
no modifier	Yes	Yes	No, unless in the same package	No

private	Yes	No	No	No
---------	-----	----	----	----

- Inheritance
 - Creates an is-a relationship between two objects
 - A well-designed inheritance hierarchy is built so
 - The parent class is the most general (contains all fields and methods common to its descendants)
 - The child classes *specialize* the parent
 - See Listings 20.1 and 20.2 in NetBeans
- Code Reuse Through Inheritance
 - If we have a group of objects, like Employees, that all have common properties and behaviors, we can put them into a base class and then extend it into particular subclasses, like Managers
 - This means we only need to write the common code one time
 - Subclasses get all the common code from the Employee class
 - Subclasses are free to add props and behaviors for their particular purposes
 - Subclasses are free to override props or behaviors of the base class
- Method Overloading
 - In any given class, each method must have a unique signature
 - We can **overload** a method by creating methods in a class that share the same name but have different parameter lists
 - See Listing 20.3
- Method Overriding
 - This means that the child class will replace the implementation of a base class method with an implementation of its own.
 - For example, all Employees objects in our system can set objectives by the `createYearlyObjectives()` method
 - But a Manager might need their own objectives AND have to set goals, so `createYearlyObjectives()`, created in the Manager class, would override (read: replace) the Employee class implementation
 - **Difference between overriding and overloading**
 - When you **override** a method in a child class, it must have the *same signature as the corresponding method in the parent class*. Overriding happens at runtime.
 - When you **overload** a method, it must have a *unique signature*. Overloading happens when compiling
 - We can have two implementations when overriding a method
 - We can replace the functionality of the parent class (we re-implement the method in the child class, which replaces the implementation in the parent class)
 - We can add to the functionality of the parent class (we re-implement the method in the child class, but at some point, we call the parent class

method)

- See Listings 20.5 and 20.6
 - The super keyword always refers to the parent of the current class
 - The this keyword always refers to the class itself
- Constructors
 - With a derived class, we have the constructor of the derived child class and the constructor of the base (parent) class
 - A derived class can extend a class that is itself a derived class
 - This means we can be dealing with several constructors
 - The first line of every constructor is either an explicit constructor call or an implicit call to super() → **the super class constructor always runs first**
 - Derived class = child class
 - Base class = parent class
 - The super keyword is useful to explicitly call the constructor of the base class or when you need to access a method of the base class that has been overridden by the derived class
 - **Rules for Dealing with Constructors in Derived Classes (Memorize):**
 - The call to super(...) must be the first statement in the constructor
 - If you do not explicitly call super in the constructor of a derived class, the compiler will automatically call the base class default constructor. If one doesn't exist, a compilation error will occur
 - The base class constructor can be invoked in a derived class by calling super(...) where (...) is the parameter that may be empty
 - You can call super(...) only from within the constructor of the derived class, not from anywhere else
 - The call to super must match the signature of a valid constructor in the base class
 - If your derived class does not define a constructor, the compiler provides the derived class with a default constructor that does nothing but call super, invoking the constructor of the base class
 - You are allowed to call this(...) as the first line instead, which will call a different constructor in the derived class
 - Eventually, the super constructor will get called (before all other constructors)
 - Calls to constructors in this() or super() must be first
 - Circular constructor references are disallowed by the compiler
 - We have access to the super keyword to access things in base classes and that by default, constructors in derived classes will call the base class constructors first
- Polymorphism
 - Applies to inheritance as well as interfaces
 - A Manager is an Employee; we can treat a Manager like an Employee...but the reverse is not true
 - Or, all dogs are mammals, but not all mammals are dogs

- Remember: derived types behave the same as base types. They can be used whenever the associated base type could be used
- Rules for Calling Methods Polymorphically
 - If you have an instance of Employee pointed to by an Employee reference, the Employee version of createObjectives is called (because no other version exists)
 - If you have an instance of Manager that has not overridden the Employee version of createObjectives and is pointed to by a Manager reference, the Employee version of createObjectives is called (again, no other version)
 - If you have an instance of Manager, but it is pointed to by an Employee reference, the Employee version of createObjectives is called (only version)
 - If you have an instance of Manager that has OVERRIDDEN the Employee version of createObjectives and is pointed to by a Manager reference, the Manager version of createObjectives is called
 - If you have an instance of Manager that has OVERRIDDEN the Employee version of createObjectives and is pointed to by an Employee reference, the Manager version of createObjectives is called
- Abstract Base Classes
 - Have some properties of regular base classes and some properties of interfaces
 - Like an interface, they can't be instantiated (only subclasses of these abstract base classes can)
 - Like an interface, can define abstract methods (definition only, no implementation) and then force subclasses to provide an implementation
 - Like a regular base class, an abstract base class can provide fully implemented methods that get inherited by child classes, aka concrete methods
 - Like a regular base class, an abstract base class can contain fields that are visible to the child classes
 - These features allow you to create classes that implement code common to potential subclasses (for reuse), force subclasses to have their own methods, and force subclasses to provide their own implementations of these behaviors
- Exercise 20.1 Refactored Code

```
public abstract class Shape {
    protected String color;

    public Shape(String color) {
        this.color = color;
    }

    public abstract int getArea();
    public abstract int getPerimeter();

    public String getColor() {
```

```
        return color;
    }
}

public class Rectangle extends Shape {
    private int height;
    private int width;

    public Rectangle(String color, int height, int width) {
        super(color);
        this.height = height;
        this.width = width;
    }

    @Override
    public int getArea() {
        return height * width;
    }

    @Override
    public int getPerimeter() {
        return 2 * (height + width);
    }
}
```

Your solution shows a good grasp of basic inheritance and class design. Here are some thoughts and suggestions for improvement:

Positives:

1. **Use of Inheritance:** You correctly use inheritance to extend the `Shape` class into specific shapes like `Square`, `Rectangle`, `Circle`, and `Triangle`.
2. **Constructors:** You provide constructors for each shape class, which is good for tracking the flow of object creation.
3. **Getter and Setter Methods:** Each class has getter and setter methods for the properties. This is good practice for encapsulation.

Critique and Suggestions:

1. **Shape Properties:**

- **color, area, and perimeter in Shape class:** These properties should be generalized for all shapes. However, `area` and `perimeter` depend on the specific shape, so you could use abstract methods for calculating them rather than having them as attributes.
- **Redundant Fields:** In the `Circle` class, you redeclare `area`, which is already in the `Shape` class. This leads to confusion. Instead of redeclaring it, you should just use the inherited field.

java
 Copy code

```
public abstract class Shape {
    protected String color;

    public abstract int getArea();
    public abstract int getPerimeter();
}
```

2.

3. Lack of Calculation Logic:

- In your current design, there's no logic to actually calculate the `area` and `perimeter` for any of the shapes. You could provide overridden methods to calculate these values based on the properties of the shapes.

For example, in `Square`, the area and perimeter are calculated based on `width` and `height`:

java
 Copy code

```
@Override
public int getArea() {
    return height * width;
}

@Override
public int getPerimeter() {
    return 2 * (height + width);
}
```

4.

5. Method Overriding:

- Consider overriding the `getArea()` and `getPerimeter()` methods in each specific shape class instead of using the setter and getter for the fields in `Shape`. This way, each shape will have its own logic for calculating these properties.

6. Redundant Code:

- `Rectangle` and `Square` have almost identical code. Since a square is a special case of a rectangle (with equal height and width), you could refactor this by either adding a constructor to `Rectangle` that enforces this or inheriting `Square` from `Rectangle`.

7. Constructor Improvement:

- Rather than just printing the name of the constructor, you could pass in initial values like color, height, and width via the constructor itself to make object creation more meaningful.

8. Encapsulation:

- Currently, you are using `protected` for many fields. `protected` allows subclasses and classes within the same package to access these fields. It would be better to make them `private` and control access solely through getters and setters to follow the principle of encapsulation more strictly.

Refactored Example:

Here's a possible refactor for the `Shape` class and the `Rectangle` class:

```
java
Copy code
public abstract class Shape {
    protected String color;

    public Shape(String color) {
        this.color = color;
    }

    public abstract int getArea();
    public abstract int getPerimeter();

    public String getColor() {
        return color;
    }
}

public class Rectangle extends Shape {
    private int height;
    private int width;

    public Rectangle(String color, int height, int width) {
```

```

        super(color);
        this.height = height;
        this.width = width;
    }

@Override
public int getArea() {
    return height * width;
}

@Override
public int getPerimeter() {
    return 2 * (height + width);
}
}

```

This approach emphasizes abstraction, encapsulation, and reduces redundancy. Overall, your solution is a great start, but introducing some more object-oriented principles (like abstraction and method overriding) would improve it further.

Lesson 21: Understanding Collections

- A collection is a data structure that groups multiple elements together into a single unit.
 - They usually represent a group of related data (such as a list of classes in a school schedule)
 - The Java Collections Framework is a library of code that allows programmers to manipulate groups of related objects so they don't have to reinvent the wheel
 - It is one API used by all Java programmers
- Javadoc
 - An HTML-formatted code document that is generated from special Javadoc tags in the comments of code
 - URL is docs.oracle.com/javase
- Collections Framework Structure
 - Consists of several interfaces and implementing classes - great real-world example of how interfaces are used in the real world. (Underlying implementations can have different characteristics i.e. Sets cannot have duplicate entries but Lists can.)
 - The documentation of an interface always includes
 - The name of the interface or class
 - Type parameters
 - Superinterfaces

- Subinterfaces
 - List of all class that implement the interface
 - Description of the interface or class
 - A note indicating the first version of Java where the interface or class was included
 - A list of related interfaces or classes
- Interface vs. Implementation
 - We program to the *interface* but instantiate using the *implementation*
 - List<String> strings = new ArrayList<>();
 - List is the interface, ArrayList is the implementation
 - The interface defines all the behavior we expect of the collection, and the implementation decides how that behavior is actually performed behind the scenes
 - Different implementations will do things sometimes differently in the background
 - The benefit is, if you change the implementation, you only need to change what instantiates it; the rest of your code can stay the same
 - Some implementations give you extra methods. For example, the Stack implementation of List gives you push() and pop()
 - You have to declare the variable with the interface; you have to declare the variable with the implementation you want to use. Example:
 - Stack<String> stringStack = new Stack<>();
- Iterator
 - An object that allows you to visit each element in a collection individually
 - All iterators have a hasNext methods that returns true if there are more elements to visit and a next method that retrieves the next element
 - If you are removing items from a collection, you typically wouldn't use the sometimes included remove method; instead, use a for loop
- Commonly Used Interfaces in the Collections Framework
 - List
 - ArrayList
 - Stack
 - Set
 - HashSet
 - TreeSet
 - Map
 - HashMap
 - TreeMap
- Differences between Lists and Sets
 - Each item in a Set must be unique; all duplicates are ignored
 - Lists allow duplicate items
 - The equivalence between objects is based on the equals and hashCode methods
 - A Set does not maintain order
 - A List **does** maintain order

- Therefore, when you retrieve items from a Set, the items may appear in a different order each time

Lesson 22: Introduction to Lists

- **List** is an interface that extends Collection. It is an ordered collection of items that may include duplicate elements
- Types of Lists
 - **ArrayList**
 - Can automatically shrink and grow dynamically, so you can easily add or remove elements as you like
 - Causes all other elements to be shifted, which is slower
 - Maintains order according to FIFO
 - **LinkedList**
 - A chained list of elements with a head and tail; each element is a node, which contains the data object and a pointer to the next node
 - A node in singly linked lists points to the next node; a node in doubly linked lists points to the next node and the previous
 - Main benefit is that insertions and deletions from the middle do not require shifting elements. You simply point the next node reference to the new node
 - **Stack**
 - Has basic methods of push() and pop() to place objects and remove them from the top, and a peek() method to examine the top object without removing it
 - Is a LIFO structure
- Understanding Generics
 - A generic type is a generic class or interface that is parameterized over types
 - Generic/parameterized types include data structures that are containers that hold other objects for us
 - It is almost always advantageous, cleaner, smaller, and easier to create containers, like Lists, that only accept a particular type
- The Angle Brackets (<>) are known as the diamond operator. When they contain a value, such as <int>, then it is called the type parameter
- Using Lists
 - Instantiating a List
 - The List class is not static, so we use the new keyword to instantiate it
 - At this point, the time of creation, you must identify the type of elements you want to store in the List
 - It is good to program interfaces - helps us to enforce encapsulation and other good practices
 - Example: List<String> strings = new ArrayList<>();
 - We create our new List to hold String objects and call it strings
 - ArrayList is a concrete implementation of List

- We do not use String within the right diamond operators because we assume it is a string based on the generic on the left hand side
- Parentheses are included because we are calling the constructor for the ArrayList, here with no parameters
- To summarize, we have now instantiated a new list of empty Strings

■ **Remember to add imports as needed or prompted**

- Lists in Action: Listing 22.1
 - The size() method on our ArrayList simply returns the number of items in our list
 - The selected type for a List cannot be a primitive type (int, boolean, char)
 - If you must, you should use the corresponding class version (wrapper reference type) *Integer* or *Boolean* or *Character* (aka wrapper classes)
 - Example: `List<Integer> numbers = new ArrayList<>();`
 - Listing 22.2
 - The add() method does what it says
 - We can also use it to place elements where we'd like in a List
 - Example: `strings.add(1, "A New String");`
 - The element is added; all elements formerly at position 1 and after have shifted
 - Listing 22.3
 - The List class is an indeed collection, which means that individual elements can be accessed via index like an array, but we use the get() method
 - For example, the element *The Second String* is at index 1
 - We can therefore also loop through the list with a for loop
 - Removing Items from a List
 - The clear() method removes all items
 - The remove() method and passing in an object will scan the List and remove it when Java finds it
 - Remove gets rid of the first matching method it finds, not multiples
 - We can also remove using an index position:
`strings.remove(1);`
- Enhanced Approaches to Access List Items
 - The Enhanced for Loop
 - Lets us visit each element in our list
`for (type Operand : List_name) {`
`}`
 Do **not remove** elements using this loop type
 - See Listing 22.5
 - Iterators

- An iterator is used to visit all the elements - and make sure it is done - in a Collection exactly once and only once
- While a List has an order, *not all Collections have an order*
- Creating an Iterator
 - An iterator is a generic type
 - `Iterator<String> iterator = strings.iterator();`
 - Must include an import!
- Using the Iterator - see Listing 22.6
 - We check to see whether an item to be read with `hasNext()`
 - `next()` will get the item on our Iterator
 - When `hasNext()` returns false, loop ends
- An enhanced for loop and an iterator basically do the same thing. The iterator breaks the work into more steps, but it is safer to use when removing items

Lesson 23: Exploring Maps

- Map Interface
 - A Map is an object that maps keys to values
 - In map, each key can map to one and only one value
 - Cannot contain duplicate keys
 - Maps are part of the Java Collections Framework, but the Map Interface itself is **not**
 - It doesn't extend Collection and is not iterable
 - We can get a Collection of either the keys or the values of a Map or a Collection of Entry objects (both Key and Value)
- HashMap
 - Instantiating a Map
 - Map is a generic type, so we must say what objects it will hold
 - In a Map, the type for the value must be provided, *as well as the type for the keys*
 - Example: storing population for countries
 - `Map<String, Integer> populations = new HashMap<>();`
 - Declares a Map that has a key that is a String value that will hold our country name and an integer that will hold our value, population
 - We are creating a reference to an interface, and HashMap is considered the concrete implementation of Map
 - Adding Data to a Map
 - We add data to a Map using the `put()` method, passing in the key first, then the value
 - See Listing 23.1
- Manipulating with Entries in a Map

- Looking Up Values in a Map
 - We use the get() method to look up a map Value, passing in the key for what we want to retrieve
 - Example: Integer foundPopulation = populations.get("Japan");
 - get() will return null if no value is found
 - Replacing Data in a Map
 - Use the put() method with a **unique** key
 - When we use put() with an existing key, therefore, we must overwrite the data
 - Removing Values from a Map - use the remove() method and pass in the key
- Getting Keys and Listing Mapped Values in a Map
 - Listing All the Keys
 - Because of the uniqueness of keys, when we retrieve them from a Map, they are placed in a Set
 - We can declare a keys Set and use the .keySet() method to return the keys
 - Example: Set<String> keys = populations.keySet();
 - We can use an enhanced for loop to cycle through each Set item that was created
 - We cannot necessarily predict the order of the keys in the Set we get using keySet(). Most Map types do not save their data in a specific order
 - Listing 23.3
 - Listing All the Values Key by Key
 - We can use the keySet() method coupled with the get() method (getKey() and getValue())
 - See Listing 23.4
 - Listing All the Values: Value Collection
 - We can use the values() method to get a Collection of all the values in our Map
 - Example: Collection<Integer> popValues = populations.values();
 - The reason we want to use a Collection instead of a Set is that while we cannot have a duplicate key, we could have a duplicate value. Collections let us have duplicates; Sets do not
 - Set Listing 23.5
 - Note: We do not have access to the key information for any of the values when we use the values() method, so it is not always an applicable method

Lesson 24: Using Simple File Input and Output

- Data Storage and Representation
 - We have two options available to use for data storage
 - We can store data in memory
 - We can store data on more permanent media such as a hard disk

- A group of related data is generally represented by an object in object-oriented languages
 - The object itself represents the thing being modeled (a student) and the fields on the object represent properties of the thing being modeled (first name, age)
 - When we permanently store data, we must decide how the data will be represented
 - The process of translating data from an object in memory into another format and writing it to permanent storage is known as marshaling
 - The reverse process, reading the data from permanent storage and translating it into objects in memory is known as unmarshaling
- File Format, Marshaling, and Unmarshaling
 - The key characteristics of any file format we choose to implement are as follows:
 - We must be able to easily tell where one record ends and the next one begins
 - We must be able to easily tell where one property within a record ends and the next begins
 - Malformed records should have little or no impact on our ability to properly read subsequent records
 - The format must make it straightforward to read from and write to the file. In other words, the format should be easy to parse
 - Given these overall requirements, for this lesson, the following file format will be used:
 - Each line in the file represents one student (satisfies bullets one through four)
 - Each field in the student record will be separated with the token ::
 - Satisfies the second item in the previous list
 - The token acts as a boundary between each field and is known as a **delimiter** (often associated with commas and tabs)
 - Marshaling and Unmarshaling Approach
 - The process for writing student data to a file is:
 - Open the file for writing
 - Go through the collection of students one by one
 - For each student, do the following:
 - Create a string consisting of student ID, first name, last name, and cohort, in that order, separated by ::
 - Write the string to the output file
 - Get the next student (if one exists) and go back to step *Create a string...*
 - If there are no more students to process, then quit
 - Close the file
 - The process for reading student data from the file
 - Open the file for reading
 - For each line in the file:

- Read the line into a string variable
 - Split the string into chunks at the :: delimiter
 - Create a new student object
 - Use the first value from the split string to set the student ID
 - Use the second value from the split string to set the student first name
 - Use the third value from the split string to set the student last name
 - Use the fourth value from the split string to set the cohort value
 - Put the newly created and initialized student object into a collection or map
 - If there are no more lines in the file, go to step *Read the line...*
 - If there are no more lines in the file, quit creating student objects
 - Close the file
- Simple File I/O
 - Writing to and Reading a File
 - We will use a PrintWriter object to write to our files (similar to writing output to the console)
 - We will pass a FileWriter object to take care of handling the writing of characters to the file
 - Both need import statements
 - See Listings 24.2 and 24.3
 - Unlike writing to the console, our items might not actually be written right away. The flush() method forces everything to be written to the file without delay
 - We must use the close() method to end the stream. Even though we don't have to manually allocate/deallocate memory in Java, not cleaning up after ourselves can cause resource leaks

Lesson 25: Applying Application Design

- CRUD Applications
 - CRUD stands for Create, Read, Update, and Delete - used to describe applications that are primarily concerned with managing, updating, and storing information
 - Used in so many business applications
- Using a Tiered Design
 - Allows us to ensure that our principles of separation of concerns, cohesion, and encapsulation are present in all parts of our apps
 - By keeping our concerns separated, we can reuse components in a variety of environments (classes and layers are not tightly coupled to each other)
- The Basic Tiers

- CRUD apps have four basic layers
 - Storage: Files, databases, or other persistent storage
 - Data access: Classes that handle retrieving and storing data
 - Business logic: Classes that handle all logic specific to your problem domain (traditionally referred to as the service layer)
 - User interface: Classes that handle all interaction with the user (traditionally referred to as the presentation layer)
- Data Transfer Objects (DTOs)
 - Used to move data from one tier to another in our apps
 - Also known as Java classes that have fields and getter/setters and no other methods

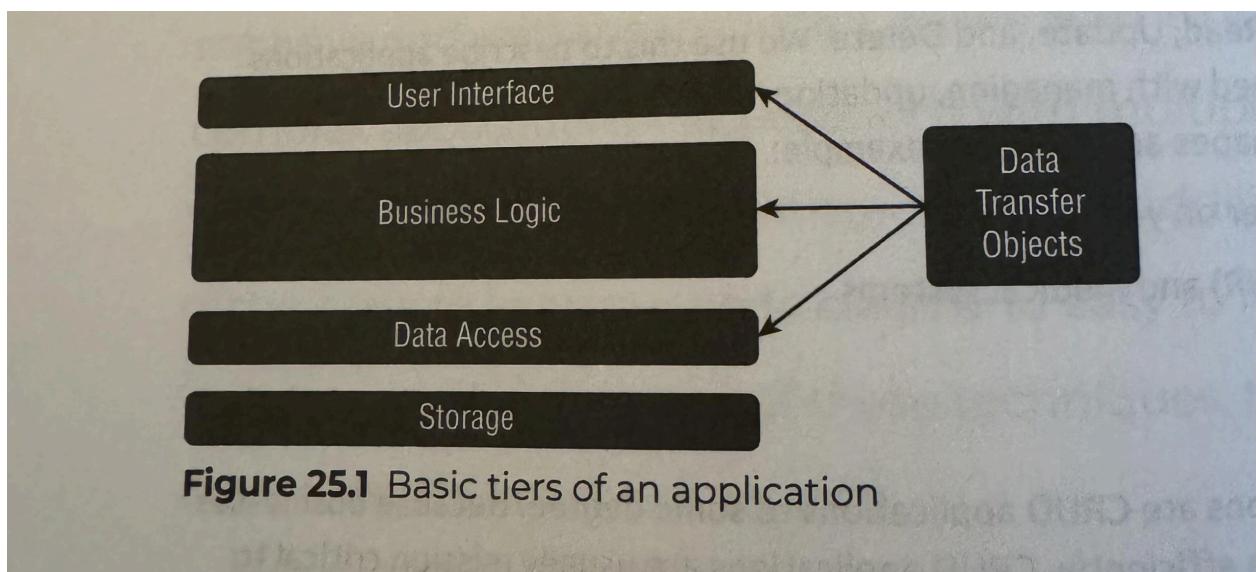


Figure 25.1 Basic tiers of an application

- The Model-View-Controller (MVC) Pattern -consists of three main layers, or types, of components
 - Model
 - Comprises all the components in the application that deal with representing and storing data
 - Includes the storage layer, the data access layer, and the DTOs
 - Controller
 - The brain of the program
 - Orchestrates the actions of the other components in the app in a way that accomplishes the app's goals
 - Think of it as a general contractor - knows the work that needs to be done, when it needs to be done, and who can do it - but never actually does the work itself
 - It uses other classes in the app to get the actual work done
 - View
 - Responsible for interacting with the user

- No other component is allowed to interact with the user

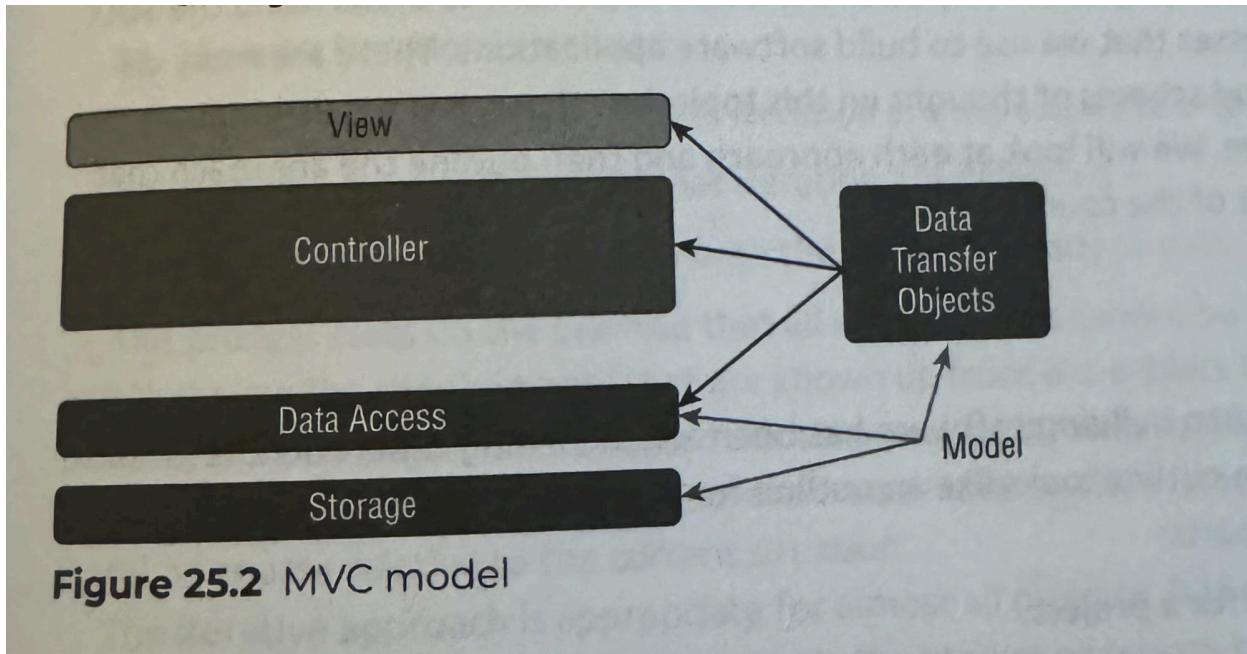


Figure 25.2 MVC model

- Design Pattern
 - A general approach or template for solving a commonly occurring problem that can be applied in many different situations
 - Can be considered best practices that developers can apply to commonly occurring software design problems
- Packages
 - Are essentially folders that help us further organize our code; we place related classes into these packages
 - Each of the basic tiers or types of classes has its own package. Recommended pattern for all projects:
 - DTOs go in the dto package
 - Data access objects (DAOs) go into the dao package
 - Controller objects go in the controller package
 - User interface objects go in the ui package
- Software Development Life Cycle (SDLC) Overview
 - Waterfall
 - The waterfall approach to building software is simple and intuitive
 - There is an idea for a project
 - All the requirements for the project are gathered, and a comprehensive specification is created
 - The software is written according to the specification
 - Testing is done against the specification
 - Bugs are reported and fixed

- The project is shipped
- This sequence is done only once - so it doesn't respond to change well! It rests on the premise that all requirements can be gathered and specified up front
- Good for systems that put people's lives on the line, like medical equipment systems. You *must* get the requirements right the first time and must be verifiable against a stable specification
- Iterative Development
 - Embraces change. Similar to waterfall steps, but the steps are repeated as many times as possible to complete the project
 - Each time through, the process adds functionality and refines the project
 - There is an idea for a project
 - Initial requirements are gathered
 - The next iteration is planned
 - The features scheduled for that iteration are developed and tested
 - The features scheduled for that iteration are delivered
 - Wash, rinse, repeat steps 2 - 5 until the project is ready to ship
 - Rests on the premise that all requirements can't be known up front and that requirements are subject to change at any point
 - Appropriate for almost all projects, especially in business. Expects changing requirements and mitigates the risks associated with those changes by delivering features a few at a time in each iteration and frequently confirms project is on course to meet client's current needs
- Agile Development
 - Project Lifecycle - taken from the OpenUP approach
 - Phases
 - Phase 1: Inception
 - Phase 2: Elaboration
 - Phase 3: Construction
 - Phase 4: Transition
 - The main idea is that as the project moves through phases on a macro level, the team is adding value and reducing risk to the project
 - Value in an agile project equates to working, production-quality code that meets the current requirements of the project
 - Phase 1: Inception
 - The beginning of any project. In order to move on, the team must:
 - Agree on scope and objectives
 - Understand what is to be built
 - Identify key system functionality
 - Determine at least one feasible solution
 - Understand the high-level estimate for cost
 - Understand the high-level schedule
 - Understand the high-level risks to the project
 - All stakeholders must agree on what will be built, the cost, timeframe, and

- feasible solution
- Phase 2: Elaboration
 - Stakeholders do the following:
 - Gather more detailed requirements
 - Design, implement, validate, and establish a baseline for the technical architecture for the project
 - Mitigate essential risks
 - Have a more accurate idea of costs and schedule
 - This phase is about gaining a better understanding of the requirements and mitigating technical risk
 - The main deliverable is an *executable architecture*, or a rudimentary working software proof-of-concept for the technically riskiest portions of the project
 - To move forward, all stakeholders must agree that the executable architecture represents the technical path forward and that the value delivered and remaining risks are acceptable
- Phase 3: Construction
 - The development team starts development of the features of the product
 - Features are delivered in an iterative manner
 - Iteration lengths are short (one to three weeks)
 - Features are delivered at the end of each iteration
 - After each iteration
 - Requirements are re-examined to make sure nothing has changed (and corrects course if necessary)
 - Plans and develops the next iteration's features
 - Testing occurs constantly and bugs are fixed before the end of the iteration
- Phase 4: Transition
 - Concentrates on getting the project ready for release
 - Includes beta testing to make sure customer expectations are met
 - Gathers information on lessons learned to improve future projects
 - Software is released; project closed
- Iterations
 - Lifecycle times
 - The project lifecycle is measured in months and keeps its focus at the overall stakeholder level
 - The iteration lifecycle is measured in weeks and focuses on the team doing the planning and work for the iteration
 - Work item level is measured in days and is focused on the individual team member
 - A few hours are spent at the beginning of each iteration planning the work to be done (Inception Phase)
 - A few days are then spent in up-front technical planning and architecture to address technical concerns (Elaboration Phase)

- After that, work of features begins; features are tested and bugs fixed. The goal is to have a stable build by the end of a week (Construction Phase)
- As the iteration ends, all completed features are finalized and put into a final stable iteration build - iteration output. A few hours are spent at the end to gather lessons learned (Transition Phase)
- These iterations continue until all the required features of the application are complete, when the project is delivered to the customer
- See Appendix C for an agile approach checklist for console CRUD applications

Lesson 26: Handling Exceptions

- Exception Handling
 - Exception is short for “exceptional event,” or something which occurs in the normal execution of a program that disrupts the normal flow of the program’s instructions.
 - There are two types of errors:
 - Runtime
 - Occur when your code is syntactically correct, but an unexpected issue occurs when you run the app
 - Example: app tries to access a file that isn’t there
 - Can be unhandled or handled
 - ComPILETIME - must be fixed before you can compile and run your app
- Catch or Specify Requirement
 - If your code can cause an exception to be thrown, you must either catch the exception or specify that your code might cause that error
 - Catching the exception means that you have written code to try to recover from the error or to simply report that the error occurred
 - Specifying the exception means marking your code to indicate that it may cause the error and letting the calling code try to handle it
- Exception Types
 - There are two categories of exceptions: checked and unchecked
 - Checked are always subject to the “catch or specify” requirement
 - Unchecked are free to handle (i.e. try to recover from) unchecked exceptions, but you are not required to
 - Anything that extends Exception (except RuntimeException) is a checked exception
 - RuntimeException extends Exception
 - Both Exception and Error extend Throwable
 - Throwable extends Object
 - Anything that extends RuntimeException is an unchecked exception
- Handling (Catching) Exceptions
 - try Block - marks the code you want to *try* to run once we identify it might cause an exception
 - catch Block
 - Each try block must generally be accompanied by at least one catch block

- Contains code that either attempts to recover from the exception or simply reports the error in some way (log file)
- If a try block contains code that throws more than one type of exception, you can have a separate catch block for each exception type so you can respond to each error in a different way
- You can determine specific exceptions to catch by reviewing Java documentation or by looking at the exceptions your program causes when it is executed
- You can combine statements if they are treated the same way using | to separate each statement
- finally Block
 - Optional code that always runs after the try-catch combination, whether an exception occurred or not
 - Well-suited for code that cleans up resources
- Overall structure of try-catch-finally

```
public static void myMethod() throws exception, another_Exception {
try {
// code here can cause an exception
// goes here
} catch (<exception_type> identifier) {
// code to handle this type
// of exception
} catch (<another_exception> identifier) {
// code to handle this type
// of exception\
} finally {
// code that runs whether an exception
// occurred or not
}
}
```

Specifying and Throwing Exceptions

- If you decide that your code should not attempt to catch the exceptions that may be thrown with the try-catch-finally construct, you must then specify that your code can cause those exceptions
 - This can be done using the throws keyword followed by a comma-delimited list, You can also specify within the body of your method, and this would instantiate a new exception
 - throws is used within the method signature, while throw is used within the method

```
public static void myMethod() throws IOException {
// method code here
```

```
}
```

```
public static void myMethod() {
    // method code can go here
    throw new IOException();
}
```

- Exception Translation and Encapsulation
 - There are times you don't want your code to attempt to recovery from an error, but you don't want to directly throw the exception that occurred either
 - Often, you want to translate the exception from an implementation-specific exception into a more general, application-specific exception. This is known as exception translation
 - Helps us maintain well-encapsulated classes and application layers
 - A leaky abstraction is when our private implementations leave errors that tie us back to them (aka the implementation details leak out to the caller through the implementation-specific exceptions)
 - To avoid this, put a try-catch block around the code that can cause the exception. In the catch block, create a new instance of the application-specific exception, passing in a message and the exception that got thrown as parameters to the constructor

```
try {
    // code that can cause an exception
    // goes here
} catch (FileNotFoundException e)
    throw new ClassRosterDaoException("Student data could not be
read",e);
}
```

Lesson 27: Building the Class Roster App

See code and notes and screenshots

Lesson 28: Exploring the Service Layer

The Role of the Service Layer

- Responsible for checking business rules, doing calculations, coordinating interactions with the DAOs, and performing other similar activities; performs **business logic**
- The MVC pattern has no place for any of this activity, which is why we need the service layer

- The controller no longer talks directly to the DAOs; instead, it communicates through the service layer, which becomes both the business logic and persistence API for the rest of the application

Service Layer Exceptions

- Represent violations of validation or business logic
- Also extend Exception and have two constructors, but they signal a different type of error

Service Layer API Design

- The first step in creating the service layer is to define what you want the service layer to do and create an interface for that functionality
- It forces us to design the API/service layer contract up front; we have to think about what it is, how it will be used, and how we signal success and failure

New Business Rules

- Define the rules
- Define any new method signatures
- Define the return type
- Define the errors that might occur in the new method(s)

Conclusion

- Application-specific exceptions are used to signal violations of business rules. These exceptions are part of the public interface, or API, of the service layer
- Service layers generally have some methods that simply turn around and call a matching DAO method, also known as pass-through methods

Lesson 30: Doing Unit Testing

- Unit Testing
 - Involves testing the lowest-level components of an application and is done by the software development team
 - The developer responsible for developing a particular component is also generally responsible for creating the unit tests for that component
 - Modern unit testing involves the use of automated testing frameworks. For Java and this course, we will use JUnit
 - As a project goes on, the entire application will be tested differently by different teams
- Types of Unit Testing
 - Black-Box vs. Glass-Box
 - Black box testing tests only the functionality of a particular component without knowing anything about how the component is implemented/ It ensures that a given component does what it promises to do
 - Glass-box/White-box testing does take the implementation of a component into account. The tests are designed so that each code path

- through the component is exercised.
- Stateful vs. Stateless Components
 - Code that has no state, or no side effects, acts like a mathematical function. If you put the same inputs into this code, it will always produce the same result
 - Code that does have state, or has side effects, yields results that might be different depending on what happened previously, even with identical inputs (i.e. entering a student with an ID of 0001 and then trying to add the same ID again)
 - We must be mindful to put things in a known state before testing stateful code, whereas we can test stateless code in any order we want
- Test-Driven Development and Red/Green/Refactor
 - TDD is a software development approach where the unit tests for a component are written before the component is implemented. The interface and shell implementation (which throw an *UnsupportedOperationException*) are the only components created before the tests are written.
 - Once the failing unit tests are written, the developer goes back and starts to replace the Exception with actual code until all unit tests pass, known as Red/Green/Refactor
 - The test results will initially all fail (Red)
 - They will turn Green as the actual code is implemented
 - Once all tests pass, the developer can refactor the code to make it more concise or efficient, and the existing tests will ensure nothing gets broken
- Test Stubs - allow us to simulate some components of the system to make the testing of other components easier
- JUnit
 - The de facto standard automated test unit that integrates with several IDEs
 - Test Setup and Breakdown
 - Hooks that run before and after your tests
 - Two flavors
 - Setup/tearDown methods that run once; useful for specialized situations
 - Setup runs before your JUnit class is created
 - tearDown runs after the test class is destroyed
 - Setup/tearDown methods that run before and after each individual test in your suite
 - More common
 - Ensures code is in a known good state before every test case is run and that everything is cleaned up when the test has completed
 - Annotations
 - The current version of JUnit has classes containing the unit test code are

- plain old Java objects (POJOs)
- Simply mark your class with JUnit annotations (i.e. `@Test`) and JUnit will do the rest
- Asserts
 - A set of static helper methods that allow you to test different conditions, known as assertions
 - Examples of assertions
 - A Boolean condition is false
 - An object reference is not null
 - Two objects are equal
 - Two values are equal
 - If the assertion is wrong, the method throws an `AssertionError`, and the test case fails
- Give/When/Then (aka Arrange/Act/Assert)
 - First, we put the code in a known good state and create all necessary test data (Given)
 - We write a test that acts; this test uses the arranged data to execute the code we are testing (When)
 - Finally, we assert that the results are what we expect (Then)
- Stateless Unit Testing
 - What Makes a Good Unit Test?
 - A good unit test must cover all the different categories of input and output combinations that are possible in a given piece of code
 - This doesn't mean you *test* every combination (impossible); rather, you must design tests that cover each class and branch of code, making a best effort not to miss any
 - Tests should be efficient but not redundant, addressing each class and branch of code uniquely
 - Designing a Test Plan
 - A test plan is a way of documenting the scope and approach of a particular group (often called a suite) of tests
 - As a whole, a test plan should encompass all types of valid inputs, as well as edge and boundary conditions, and occasionally invalid potential inputs
 - See Listing 30.1
 - Again, with stateless code, we design tests for a method without knowing anything about the methods internal code, As long as we known the *expected* behavior of a method, we can design and implement tests
 - Designing a test plan for a method means we have to decide on values for its input parameters and expected return value as a result of using those input parameters
 - **Insert picture from bottom of p. 567**
- Implementing Unit Tests
 - To create a test class in NetBeans

- Right-click the class you want to test
 - Select Tools > Create/Update Tests
 - Walk through the wizard
 - Ensure that *Test Packages* is selected in the Location dropdown
 - Generated Code Checkbox List should have nothing selected
 - Method Access Levels Checkbox List should have nothing selected
 - Generated Comments Checkbox List should have nothing selected
- Reminders
 - All tests *must* have an @Test notation or JUnit won't know to run it as part of the test suite
 - Test methods should always be public, return no value, and have an empty parameter list. Only the names of our test methods will change
 - Name your test cases so that you can tell what they are testing (less of a requirement than a good idea)
 - Each test method should test only one input/output combination - makes it easier to debug your tests when they fail
 - Use the assertXxxx methods to determine whether the test results are what you expected (i.e. if you expect the first test to return false, name the method assertFalse)
- Running JUnit Tests
 - Do a Clean and Build
 - Right-click your project in project view and select *Test* from the dropdown
 - To easily reopen the Test Results window, navigate to Window > IDE Tools > Test Results
 - To view the passed tests, click the check mark in the green circle on the left to show them

Lesson 31: Testing Stateful Code

- Unit Testing Stateful Code
 - You must put the system in a known good state before testing stateful code. Otherwise, you will have no way to check to see whether the actual result of the test matches your expected result (not necessary for stateless code)
 - You must be mindful of the order in which you test things in stateful code. Calling the same method with the same parameters three times in a row may produce three different results if the code is stateful
 - The expected results from a particular method call might depend on if and/or how many times a different method was called in stateful code. In other words, calling one method can change the result of calling another method, which doesn't happen with stateless code
- Arrange, Act, Assert - you can do this as many times as needed to fully execute the test case
- A stubbed version of a component simply returns canned data for any method call. We can set up a stubbed version of a component to act just about any way we want or need

it to.

Lesson 32: Enums

- Magic Number
 - A value that is hard-coded into your code but does not have a clear meaning
 - For example:

```
public void shipOrder(Order order) {  
    if (order.getStatus() == 2) { // purchased  
        // ship it  
  
        // move to shipped status  
        o.setStatus(3);  
    }  
}
```

- But what do those numbers mean? The comments tell us that they are order statuses. But what if there needs to be a change in the order stages/steps?
- Enum
 - A construct that allows us to define a group of related constants, or a way to define a controlled vocabulary for the constants we are interested in
 - An example is to use an enum to define the order status options in the previous example or define days in the week
 - Enums take constants further in that they are all grouped together and are type safe
 - Think of an enum as a better constant for now
- Creating Enums for Fixed Constants
 - Similar to defining a class or interface
 - List each item in the enum vocabulary in a comma-separated list in the body of an enum
 - Name them in all caps
 - Example

```
public enum MathOperator {  
    ADD, SUBTRACT, MULTIPLY, DIVIDE  
}
```

- Using Enums
 - An enum is a proper type, which means we can use it in the formal parameter list of a method and in switch statements
 - See Listing 32.2
- Getting Values from an Enum

- We might use an enum defined in a different file
 - When an enum is created by the compiler, a method called values() is added to the enum, which allows you to access the values the enum contains
 - See Listing 32.4
- Enum Members
 - Members of an enum are implicitly declared as public static members that cannot be changed
 - See Listing 32.5

Lesson 33: Manipulating Dates and Times

- ISO Calendar
 - The Java Date-Time API uses standard 8601 from the International Organization for Standardization (ISO 8601)
 - Meant to normalize the differences in date/time formats across the globe and is based on the Gregorian Calendar, introduced in 1582
 - Basic format: YYYY-MM-DDThh:mm:ss
 - Time is represented using the 24 hour clock
- Human Time and Machine Time
 - Human time is regular calendar and clock time, and consists of units such as years, months, days, hours, minutes, and seconds
 - Machine time is a timeline, measured down to nanosecond intervals, that starts on January 1, 1970, which is known as the epoch
 - Positive numbers represent time after the epoch
 - Negative numbers represent time before it
- Local and Zoned Classes
 - The Java Date-Time API provides classes that work with time zone info, and others that deal with dates and times without taking time zones into account
 - *ZonedDateTime* takes the time zone into account
 - *LocalDateTime* does not
- Periods and Duration
 - Both are used by the Java Date-Time API (JDTAPI) to measure time periods
 - Durations are measured in machine-time (seconds and nanoseconds)
 - Periods are measured in human time
- Working with LocalDate OBJECTS
 - Creating Local Dates
 - Created using one of the supplied factory methods rather than using the *LocalDate* constructor
 - A factory method is simply a method that creates a new object for us when called
 - Most common methods are now and parse
 - See Listing 33.1 and Listing 33.2
- Converting Dates to and from Strings
 - We often get dates from a file not stored in a *LocalDate* format but rather

- presented as a simple string of text
- See `parse()` in Listing 33.3
- Formatting Dates
 - To display dates in a variety of formats, we use the `LocalDate.format()` method and the `DateTimeFormatter` class
 - The following also formats a `LocalDate` into a `String` with the date pattern `mm/dd/yyyy`

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy");
LocalDate ld = LocalDate.parse("02/07/2021", formatter);
String formatted = ld.format(formatter);
```

- The `DateTimeFormatter` is being created, called `formatter`
- The `DateTimeFormatter` is then passed to the `parse()` method of `LocalDate` along with a date value to create a `LocalDate` in the second line
- We then assign the value returned from calling the `format()` method on our `LocalDate` object (`ld`) to our formatted `String`, `formatted`
- See Listing 33.4 for another example
- Using Localization
 - The `ofLocalized-Date()` method uses localization info from the system to determine how the date should be formatted
 - The format for usage is:
`ld.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL));`
 - The predefined style is a `FormatStyle.FULL` **enum**
 - See Listing 33.5
- Getting the Time with `LocalDateTime`
 - You can get both Date and Time with a `LocalDateTime` object
 - See Listing 33.6
- Working with Date Calculations
 - We can manipulate dates with `LocalDate`
 - Calculating Dates in the Future and Past
 - Some methods included that let you calculate dates
 - `plusYears()`
 - `plusMonths()`
 - `plusWeeks()`
 - `plusDays()`
 - `minusYears()`
 - `minusMonths()`
 - `minusWeeks()`
 - `minusDays()`
 - To use these methods, pass in the number of units you want to add or subtract from the existing date, and the method will return a new

LocalDate object

- See Listing 33.7

- Calculating Time Between Two Dates

- The until() method compares two dates and returns a Period object representing the difference in time between the two dates

- See Listing 33.8

- Working with Legacy Dates

- Converting Date Objects

- In Java, we may need to convert legacy Date and GregorianCalendar objects to LocalDate objects

- To convert from legacy Date to LocalDate object consists of two steps

- Convert the Date into a ZonedDateTime object

- Convert the Date into an *Instant* (from human time into machine time)

- We then convert the Instant derived from the legacy Date into a ZonedDateTime object using the static ofInstant() method

- We essentially convert the machine time Instant back into a human time ZonedDateTime object

- We must also pass a time zone ID into the ofInstant() method

- We do this by using the system default of the machine the code is running on

- Call the toLocalDate() method of our ZonedDateTime object

- See Listing 33.10

- Converting a GregorianCalendar Object

- Much easier: GregorianCalendar includes a method to convert to a ZonedDateTime object called *toZoneDateTime()*

- Then, follow the second step above to convert the ZonedDateTime to a LocalDate

- See Listing 33.11

Lesson 34: Using the BigDecimal Class

- Exploring BigDecimal

- Used to represent arbitrary precision decimal numbers
- Provides developers with a way to deal with decimal numbers in a predictable fashion
- Provides ways to set the number of significant digits and configure how numbers should be rounded
- Should be used for all calculations involving currency
- It is an immutable type, meaning a BigDecimal object's value cannot be changed once it is set, and you must set the result of any BigDecimal operation to another BigDecimal variable

- Constructing BigDecimals
 - You can use a *double* constructor to create a BigDecimal, but it gets stored all crazy and isn't recommended. See Listing 34.1
 - Ironically, a String constructor would be better to create BigDecimals with an exact value. See Listing 34.2
 - None of the other BigDecimal constructors (int, long, BigDecimal) have the problem of imprecision
- Understanding Scale - the scale value is the number of digits to the right of the decimal point (i.e. currency calculations have a scale value of 2)
- Understanding Rounding Modes
 - The rounding mode you use will be dependent on the business rules for your application - should *not* be made by the developers
 - See Table 34.1:

Table 34.1 BigDecimal Rounding Modes

Rounding Mode	Description
CEILING	Rounds toward positive infinity
DOWN	Rounds toward zero
FLOOR	Rounds toward negative infinity
HALF_DOWN	Rounds toward the nearest neighbor unless both neighbors are equidistant, in which case it rounds down
HALF_EVEN	Rounds toward the nearest neighbor unless both neighbors are equidistant, in which case it rounds toward the even neighbor
HALF_UP	Rounds toward the nearest neighbor unless both neighbors are equidistant, in which case it rounds up
UNNECESSARY	Asserts that the requested operation has an exact result so there is no need to round
UP	Rounds away from zero

- You can import `java.math.RoundingMode` to get the `RoundingMode` enum values
- Working With BigDecimal: Setting Scale - Listing 34.3 and 34.4
- Doing Calculations with BigDecimal - Listing 34.5
- Dividing BigDecimals
 - When we divide, it is possible for decimal places to increase i.e. $10 / 6$
 - The result is 1.6666 - a nonterminating decimal that doesn't end and that BigDecimal doesn't know how to store
 - See Listing 34.6

Lesson 35: Lambdas and Streams

- Using Aggregate Operations
 - Use *lambdas* to perform actions on the objects in a Collection, which are used to store objects in our program
 - Actions include storing, retrieving, removing, and updating the objects

- Examples:
 - Print the names of all people in a Collection of Address objects
 - Return all of the Address objects for people in Akron, Ohio
- Understanding Pipelines and Streams
 - A pipeline is a sequence of aggregate operations. Include a data source, zero or more intermediate operations, and a terminal operation
 - Data source
 - Most commonly a Collection
 - Could also be an array, return from a method call, or some sort of I/O channel
 - Intermediate operations (i.e. filter operation)
 - Accept a stream and produce a new stream
 - For instance, a filter operation takes in a stream and then produces another stream that only contains the items matching the criteria of the filter
 - Terminal operation
 - Operation that returns a nonstream result
 - Could be a primitive type (i.e. an integer), a Collection, or no result at all (i.e. it just prints the name of each item in the stream)
 - *A data source and a terminal operation are required for the pipeline to execute*
 - A stream is a sequence of items that carries items from the source through the pipeline
- Streams vs. Iteration
 - Aggregate operations process items from a stream, not directly from a Collection
 - Aggregate operations support lambda expression as parameters
 - Anything we do with streams can be accomplished by using normal loops and iteration, but it might take a lot more code
- Exploring Lambdas
 - In programming, lambdas are anonymous functions or methods
 - Anonymous means we don't define them with a specific name
 - They still take in parameters, have a body, and can return data
 - In Java, lambda expressions are not anonymous; they are implementations of functional interfaces
 - A functional interface is simply an interface with a single abstract method
 - I.e. the Runnable or Comparable interface
 - Lambda expressions can be passed into methods as a parameter, which essentially allows us to pass methods into methods
 - They are particularly useful with the stream and aggregate operation features of Java. Aggregate operations (filter, forEach) require the caller to pass in a lambda that defines how the objects in the stream should be filtered or processed
 - In Java, lambda expressions allow us to treat code as data
- Working with Stream and Lambda Syntax
 - See Listing 35.1
 - The forEach() Stream Method

- Returns nothing; allows us to run code against each and every object in the stream
- The lambda passed in this example implements the functional interface Consumer, which includes the abstract method **accept**
- One way to use this is to print everything in a list
- See Listing 35.2
- Listing 35.3 shows how to combine the forEach() and filter() methods
- The filter() Stream Method
 - Filters a stream down to contain only the objects we want it to
 - The lambda we define for it is a Boolean check: anything that returns true from the check stays in the stream; everything else is discarded from the stream
 - Example Code

```
// Assume we are starting from a List of Person objects called people
// We use the stream method on the List to turn it into a stream, then we
call the filter method on that

people.stream().
    filter((p) -> p.getAge() >= 18)

// Our lambda is
(p) -> p.getAge() >= 18

// p is the parameter of the lambda; can be called anything
// It will hold each person object as we evaluate it in the stream
// The arrow operator, ->, is used only for lambdas to separate the
parameters from the body of the lambda
// The body is p.getAge() >= 18

// We can also chain stream method calls
// We first filter down to people over the age of 18
// and then filter down to people whose names start with J
people.stream()
    .filter((p) -> p.getAge() >= 18
    .filter((p) -> p.getName().startsWith("J"))
```

- Note
 - A lambda that returns true or false in Java can also be called a *predicate*
 - The lambdas passed into filter operations are predicate implementations

- Predicates take one parameter and return a Boolean primitive
- Other types of predicates include BiPredicate, which takes two parameters
 - IntPredicate (takes in an int parameter)
 - LongPredicate (takes a long parameter)
- The map() Stream Method
 - Allows us to switch the stream to contain only the data that can sometimes be easier to process
 - We use this when we want to convert the type that is being held in the stream
 - Different from the Map class
 - The map method takes a function as a parameter
 - The input function is a functional interface that takes as input a generic parameter and returns a generic value

```

people.stream()
    .map((p) -> p.getName())

// We turn our List into a stream with the stream() method
// then we call our map() method

// Inside map(), we use a lambda to return a piece of data with its type
// being the type of the stream
// In this case, by returning the name of the Person object, the stream is
// now a stream of String objects, instead of a stream of Person objects

// The lambdas now take in Strings

// We can use special map methods to map into ints, doubles, or longs
// mapToInt(), mapToDouble(), mapToLong()
// They create IntStream, DoubleStream, and LongStream that let us
// use methods such as average(), sum(), min(), or max()

// Example - get the average age of all Person objects in our list
int averageAge = people.stream()
    .mapToInt((p) -> p.getAge())
    .average();

// map() can be combined with any other stream method

people.stream()
    .filter((p) -> p.getAge() >= 18)
    .map((p) -> p.getName())

```

- The collect() Stream Method
 - filter() and map() return a stream
 - The collect() method allows us to change the stream back into a List to pass it around and use it in our code properly
 - We pass into it a Collector type, which we can generate using static methods into a List
 - See Listing 35.4
 - We can also collect our stream into a Map

```
Map<Integer, List<Person>> peopleAges = people.stream()
    .collect(Collectors.groupingBy((p) -> p.getAge()));

// In the call Collectors.groupingBy((p) -> p.getAge())
// we specify what we want the key for the Map to be
// and how we're organizing our List; in this case, by Age
// For each distinct age, we will have a List of Person objects
```

- You can find more things to do with streams, take a look at the Stream Javadoc

Lesson 36: Working with the Spring Framework

- Introduction
 - **Spring** is a collection of libraries that provide support for JVM-based enterprise applications
 - The Spring framework provides support for dependency injection, MVC web applications, RESTful web services, authentication/authorization, and database connectivity
- Dependency Injection and Programming to Interfaces
 - Review: Dependency injection is a design pattern that implements a form of inversion of control (interchangeable terms)
 - In this case, the inversion is that client objects are no longer responsible for instantiating the objects (services) on which they depend
 - Instead, the dependencies are handed to/injected into the client objects by some other entity (constructors or setter methods)
 - Before this chapter, everything was hard-coded into the App class, and we would have to recompile it every time we made changes
 - The Spring framework allows us to externalize this configuration into XML files or use annotations to define our DI
 - Advantages of Dependency Injection
 - Allows for loose coupling between the client and the concrete implementation of the service

- Allows the externalization (to configuration) files of the system's configuration information. This allows for configuration changes without forcing a recompilation of the app
 - Allows for more flexible parallel development. Developers can program against the interface and use stubbed or mock implementations while the real implementation of the component is being built.
- Why Spring?
 - Spring was originally created as a reaction against the growing complexity of the Java frameworks that existed at the time (around 2002 - 2004)
 - Its sole purpose is to simplify the programming and creation of Java applications
 - The base feature of dependency injection because Spring's aforementioned features (MVC, web services, etc.) build on this foundation
 - Spring Boot, included with one of Spring's more recent releases, enables automatic configuration based on what libraries are packaged with or available to your application
- Understanding the Spring Approach
 - Plain Old Java Objects
 - Many frameworks require you to extend their classes to take advantage of their features; you can end up being locked into a particular vendor's solution
 - Spring aims to be minimally invasive in that it allows you to use POJO's instead
 - POJO's can be largely free of framework-specific code
 - Makes it more testable (can be tested outside the framework)
 - Easier to move to another framework if desired
 - The Spring container uses the POJO classes and the configuration data to configure and run an app. Provides two containers that support DI
 - BeanFactory container: The simplest container with basic support for DI
 - ApplicationContext container: Built on top of the BeanFactory to provide more enterprise-specific functions. Recommended to use this over BeanFactory because it contains all the features of BF
 - All the objects managed by the Spring container are called beans. The Spring container is responsible for instantiating, assembling, and managing objects (also called beans).
 - The container uses configuration metadata to know which objects to instantiate, assemble, and configure. We can use XML to represent the configuration data
 - Templates
 - Spring uses templates to reduce the need for boilerplate code in your apps
 - See the difference between Listing 36.1 and Listing 36.2

LISTING 36.1

No Spring

```
public Employee getEmployeeById(long id) {  
    Connection conn = null;  
    PreparedStatement stmt = null;
```

```
ResultSet rs = null;
try {
    conn = dataSource.getConnection();
    stmt = conn.prepareStatement(
        "select id, firstname, lastname, salary from "
        + "employee where id=?"); //select employee
    stmt.setLong(1, id);
    rs = stmt.executeQuery();
    Employee employee = null;
    if (rs.next()) { //this will create an object from the data
        employee = new Employee();
        employee.setId(rs.getLong("id"));
        employee.setFirstName(rs.getString("firstname"));
        employee.setLastName(rs.getString("lastname"));
        employee.setSalary(rs.getBigDecimal("salary"));
    }
    return employee;
} catch (SQLException e) { // what should be done here?

} finally {
    if (rs != null) { //clean up mess
        try {
            rs.close();
            catch(SQLException e) {}
        }

        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException e) {
            }
        }
    }

    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {
        }
    }
}

return null;
}
```

LISTING 36.2

With Spring

```
public Employee getEmployeeById(long id) {
    return jdbcTemplate.queryForObject (
        "select id, firstname, lastname, salary " + // SQL Query
        "from employee where id=?",
        new RowMapper<Employee>() {
            Public Employee mapRow(ResultSet rs, //Map results to object
            int rowNum) throws SQLException {
                Employee employee = new Employee();
                employee.setId(rs.getLong("id"));
                employee.setFirstName(rs.getString("firstname"));
                employee.setLastName(rs.getString("lastname"));
                employee.setSalary(rs.getBigDecimal("salary"));
                return employee;
            }
        },
        id); //specify query parameter
}
```

- Spring templates allow the developer to concentrate on the business logic instead of repetition resource management and error handling code
- Class Annotations
 - Each class that was a bean in the XML setup needs to be annotated; the annotations are how Spring figures out which classes to instantiate
 - Add the @Component annotation before the class declaration
 - This is the primary annotation we use to tell Spring to instantiate this class in memory so it can be injected elsewhere
 - It comes from the Spring library we added, so it will need to be imported
 - We only deal with the Impl versions of classes, so we do not need to annotate any interfaces
 - The classes in Listing 36.7 have no dependencies, so this is all we have to do
 - For classes with actual dependencies
 - We also add the @Autowired annotation before the constructor (in addition to the Component annotation before the class declaration)
 - This tells Spring that when a BookView is created, it should look for something that is-a UserIO in memory to inject
 - Since we annotated the UserIOConsoleImpl class, that class will be in memory and available to inject
 - Another way to do this would be to put the @Autowired annotation directly on the io field if we didn't have the constructor

```
@Component  
public class BookView {  
  
    @Autowired  
    private UserIO io;
```

- Using constructor autowiring will avoid null resources at runtime. If Spring can't find a bean to inject, the context load will fail if using the constructor. On the field, you could get a null value
- See Listing 36.9 for annotations for BookService and BookController
- See our modified main method in Listing 36.10
- In this annotated version, the appContext is an AnnotationConfigApplicationContext class, which will need to be imported from the Spring library
 - Once we have this instantiated, we need to tell Spring where to start scanning our project for annotations
 - We typically want to start scanning from the package App is in, in this case, com.sg.booktracker
 - When we call the refresh method on the next line, Spring checks all classes it can find in that package and any child packages for annotations
 - Anything that is an @Component is created in memory with dependencies injected where it sees @Autowired
 - We can then ask for our BookController so we can start the program
 - Since we didn't set any IDs for our classes when we annotated them, their default IDs are their names converted to camel case, in this situation bookController for BookController
- Summary
 - Lesson 36 covers what the Spring framework is and how we can use it for dependency injection
 - We have to add our external libraries as dependencies to our Maven POM file
 - The Spring application context can be set up using XML or annotations
 - To use the Spring application context, we need to read in the XML file or scan for annotations
 - Annotations are the more common option in Java development these days

Lesson 37: Introducing Maven

- What is Maven?
 - It is a project management framework.
 - It strives to manage a project's build, reporting, and documentation from one

- place
 - It's declarative rather than task-oriented
 - Contains a built-in lifecycle so you can declare what you want to do, not how to do it
 - Has declarative dependency management, so you tell Maven what libraries and version numbers you need, and it will make them available
- Project Object Model
 - Maven is based on the Project Object Model (POM), which is defined in a **pom.xml** file
 - This file contains the declarations for all libraries on which the project depends and can contain declarations of the Java version to use and other project-level settings
 - See Listing 37.1
- Project Identification
 - The lines of code from *groupId* to *name*
 - groupId: Traditionally, it is the base package of your project, but this is not a requirement. The groupId is meant to identify the organization with which the project is associated
 - artifactId: The name that will be given to the file into which the project is packaged
 - SNAPSHOT: The snapshot indicates the version of your project. It defaults to 1.0-SNAPSHOT in NetBeans, but you can set this to any value
 - packaging: The packaging tag indicates how you would like the project to be packaged. For now, this will always be jar, which stands for Java Archive. If you were doing web applications, then this would be war, which stands for *web archive*
 - name: The name tag is, essentially, the name of the project. Does not have to match the ArtifactId
- Dependencies
 - Wrapped by *dependencies* tags
 - Each library on which we depend has a separate dependency tag nested in the *dependencies* tag
 - Use many of the same tags used in project identification: groupId, artifactId, and version
 - The scope tag can be used to limit where and how a particular library will be used. In the example, it means JUnit should only be used in tests but not in packaging or installing the project
- Build Settings
 - Contained within the build tags
 - With the build tags, we are interested in the target and source tags of the compiler plugin.

- The source tag indicates the Java version that should be used for the source code of the project, and the target tag indicates the version of the JVM on which the project should be run
- Dependency Management
 - One of Maven's great features; it means that Maven will manage all the external libraries (packaged as JAR files) that you need to use in your program.
 - The word *dependency* is used differently between Spring and Maven
 - Spring helps us manage how the components of our application relate to and depend on each other at the class and object levels through dependency injection
 - Maven helps us manage the external Java libraries that we rely on at a project level
 - Without Maven, the developer is responsible for manually identifying, downloading, and including the JAR files of all the libraries on which the project depends, as well as storing and managing the JAR files
 - With Maven, the developer is still responsible for identifying the libraries on which the project depends, but Maven automatically fetches all dependencies into a central repository on your machine (located in the `~/.m2` directory). All Maven projects on your machine share this repo, which means that each library is downloaded only once
 - Maven also handles *transitive dependencies* automatically
 - If you declare in your POM that Library A depends on Libraries B and C, Maven will automatically download all three libraries into your local repository
 - You don't have to specify or be aware that A requires B and C
- Maven Lifecycle
 - The Maven lifecycle consists of several stages, called goals, or kinds of actions (like compiling or running unit tests) that we need to take as we build a software project
 - The lifecycle is flexible, but most developer use the standard defined goals:
 - **compile:** Compiles the project source code
 - **test-compile:** Compiles the project test source code
 - **test:** Runs the project unit tests
 - **package:** Builds and packages the project
 - **install:** Installs the project package into the local `.m2` repository (the project package can then be used in other places)

Lesson 38: Pulling it All Together: Building the Class Roster with Spring

- Introduction
 - We will have Spring do the dependency injection and application wiring that we currently do by hand in the App class and the service layer unit test
 - Using the Spring DI container is a foundational Spring skill and is used in just about every Spring project you encounter

- Include the Spring Libraries - ensure Spring is included in the Maven POM, with the proper dependencies
- How Do I Know What My Maven Dependency Entry Should Be?
 - Go to the project website (i.e. the Maven entry for the Spring core framework is found on Spring Boot's Maven Plugin webpage)
 - Search for the dependency on Google or another search engine (i.e. JUnit maven dependency)
- Add Spring Configuration Files
 - Add the *application context* configuration files
 - One way to do this is through an XML file called `applicationContext.xml`, as defined in Listing 38.2
 - You can use Listing 38.2 as a starting point for all your Spring configuration files
 - You should also create a separate xml file for testing, to be manually created in `src/test/resources`
- Convert the App Class to Use Spring (Dependency Injection - Listing 38.3)
 - We're going to move and convert the current code to fit `applicationContext.xml`
 - Then we'll replace the code currently in App with code that instantiates the Spring DI container
 - Then we'll calls run on the controller
- Defining Beans
 - Before injecting Spring, our App class is responsible for instantiating each of the objects in our application, which requires knowing the dependencies between the components and passing the correct objects into appropriate constructors when instantiating the classes
 - Spring allows us to remove this knowledge from our Java code and externalize it to the `applicationContext.xml` file
 - We'll define the relationships in `applicationContext.xml`
 - The Spring DI container will handle the instantiation of our objects
 - Beans tags and attributes
 - **The bean tag:** The XML tag used to define the objects we want the Spring container to instantiate for us
 - **The id attribute:** The XML attribute that we use to tell the Spring container what alias to give a particular bean that it instantiates. It is similar to the key in a Map. we can ask for a reference to the object by passing the id attribute to the Spring application context, and it will hand us back the requested reference
 - **The class attribute:** This is the fully qualified name of the class that we want the Spring container to instantiate for us. Remember, this must be the fully qualified name of the class. You must explicitly include the full package name.
 - **The constructor-arg tag:** This tag is nested in the bean tag and is used to indicate that we want Spring to pass a parameter into the constructor when instantiating the bean

- **The ref attribute:** We use this attribute to refer to a previously defined bean. The value of the ref attribute must be the id of another bean in the applicationContext.xml file
 - See Listing 38.4
- Exception Conditions
 - One consequence of externalizing dependencies to an XML file is that any errors contained in the configuration file are not detectable until runtime, because the compiler has no way of catching these errors
 - Common Errors
 - Misspelled bean class attribute
 - All bean classes must be specified by their correctly spelled, fully qualified class name
 - The Spring container will throw an error indicating that the bean cannot be instantiated
 - Misspelled bean id in ref attribute: When you refer to another bean (in a ref attribute), it must appear exactly as it did in the original bean definition
 - Malformed XML
 - The applicationContext.xml file must be composed of well-formed XML
 - These are the easiest errors to catch and diagnose, as it usually tells you the error's line and column
 - Using the value attribute when you mean to use the ref attribute in constructor or setter injection
 - Remember, the value attribute passes the literal value of the attribute to the constructor or setter
 - The ref passes a reference to the object referred to by the object value
 - You can also provide configuration metadata for the Spring container through the use of annotations or Java code

LISTING 31.1

The New ClassRosterDaoFileImpl Constructors

```
public ClassRosterDaoFileImpl(){  
    roster_file = "roster.txt";  
}  
  
public ClassRosterDaoFileImpl(String rosterTextFile){  
    roster_file = rosterTextFile;  
}
```

This has created a pair of constructors. The first, no-args constructor is providing the earlier default behavior that `ClassRosterDaoFileImpl` was originally built upon, which is instantiation and the assignment of `roster.txt` to the `roster_file` variable. However, the second, overloaded constructor has allowed us to create `ClassRosterDaoFileImpl` instances that use another file, thus allowing the file reference to be injected upon construction. This is something that will be perfect for test setup and ensuring that we don't overwrite our production application data.

Adding hashCode and equals to Student

To make testing easier, we implement the `equals` and `hashCode` methods for the `Student` object. This will allow us to easily compare two `Student` objects to see if the values of their fields match.

Both `equals` and `hashCode` are methods inherited from the `Object` class. The default implementation of `equals` simply compares the heap location of two `Student` references to see if they are pointing to the same place on the heap, equating whether or not they are literally the same object. While useful, during testing it would be more useful to have the method work similarly to the `String`'s overloaded version, where when comparing two Students' contents, if the same, those two `Student` objects would be considered equal.

We will only use the `equals` method for our tests. This requires that if two objects are equal, then the `hashCode` values for each object must also match. As such, we must also implement the `hashCode` method. We won't get into all the details here, but essentially, the `hashCode` of an object is a unit integer value that represents the state of that object. In short, two objects that are equal to one another must have the same `hashCode` value.

Luckily, NetBeans (and every other IDE) will generate these methods for us and will help ensure that they are implemented correctly. In fact, NetBeans won't let you generate one without the other.

IMPORTANT NOTE

If you override either `equals` or `hashCode`, you are expected to override the other, and you should use the same properties in both. Failure to do so can cause irregular or breaking issues in Java code.

Simply follow these steps to add `equals` and `hashCode`:

1. Open the Student class and right-click the class name to get a pop-up menu. Select the *Insert Code* option in the menu.
2. Select `equals()` and `hashCode()` from the list of options.
3. Select all four checkboxes in both sections. All the Student properties are important for this.
4. Click the *Generate* button.

You should end up with two methods that look similar to the ones in Listing 31.2. Make sure all the properties are accounted for in yours, but don't worry if the numbers are a little different. The numbers are autogenerated, and it is not as important that they match exactly with the ones in our example.

LISTING 31.2

Student's New `equals` and `hashCode` Methods

```
@Override  
public int hashCode() {  
    int hash = 7;  
    hash = 89 * hash + Objects.hashCode(this.firstName);  
    hash = 89 * hash + Objects.hashCode(this.lastName);  
    hash = 89 * hash + Objects.hashCode(this.studentId);  
    hash = 89 * hash + Objects.hashCode(this.cohort);  
    return hash;  
}  
  
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj == null) {  
        return false;
```

```
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Student other = (Student) obj;
        if (!Objects.equals(this.firstName, other.firstName)) {
            return false;
        }
        if (!Objects.equals(this.lastName, other.lastName)) {
            return false;
        }
        if (!Objects.equals(this.studentId, other.studentId)) {
            return false;
        }
        if (!Objects.equals(this.cohort, other.cohort)) {
            return false;
        }
    }
    return true;
}
```

Now that you have these methods, you can assert on whole Student objects to check their equality with another Student object.

Adding `toString` to Student

It is also recommended to add an overridden `toString` method to your `Student` class to help with test failure messages. This `toString` method is added mostly for convenience. Often in error messages, JUnit will print out information about the object that failed a test. It helps if that information is readable to us, but the default `toString` method only really serializes the object's class name and hashcode. Both are interesting pieces of information, but not often particularly usable. Overriding this method can allow us to print out all the object's property values instead, which can allow for much better insight into issues when reading test logs.

To add your new `toString` method, follow the same process we used earlier for adding the `equals` and `hashCode` methods. Right-click the `Student` class name and select `Insert Code`. From the menu, select `toString` to display the code generation dialog. From the dialog, select all the fields and then click the `Generate` button to add the code that will be similar to Listing 31.3.

LISTING 31.3

Example of Student's New `toString` Method

```
override  
public String toString() {  
    return "Student{" + "firstName=" + firstName + ", lastName=" + lastName +  
    ", studentId=" + studentId + ", cohort=" + cohort + '}';  
}
```

Creating the Test Class

Our next step is creating the JUnit test class that will hold our test suite for our DAO. To do this, we follow similar steps to the ones done in the stateless testing exercises.

1. In NetBeans's Project view, right-click the `ClassRosterDaoFileImpl` class.
2. Select *Tools*, and then select *Create/Update Tests*.
3. Walk through the wizard to create a new JUnit class.
 - The name should look similar to `classroster.dao.ClassRosterDaoFileImplTest`.
 - Ensure that *Test Packages* is selected in the Location drop-down.
 - Generated Code Checkbox List should have everything selected.
 - Method Access Levels Checkbox List should have nothing selected.
 - Generated Comments Checkbox List should have nothing selected.
4. Click *OK*.

Once you complete the wizard, you should be left with a new Java class similar to Listing 31.4. This class is the shell of our new test suite, and we will be creating new tests to replace the `testSomeMethod` method with real tests.

LISTING 31.4

`ClassRosterDaoFileImplTest`

```
public class ClassRosterDaoFileImplTest {  
  
    public ClassRosterDaoFileImplTest() {  
    }  
}
```

```
@BeforeAll  
public static void setUpClass() {  
}  
  
@AfterAll  
public static void tearDownClass() {  
}  
  
@BeforeEach  
public void setUp() {  
}  
  
@AfterEach  
public void tearDown() {  
}  
  
@Test  
public void testSomeMethod() {  
    fail("The test case is a prototype.");  
}  
}
```

The Set Up and Tear Down Methods

The JUnit Framework gives us four methods, or rather four annotations, to help get the code we are testing into a known good state.

- `setUpClass` annotated with `@BeforeAll`
 - This static method is run once at the time the test class is initialized and can be used to set up external resources.
- `tearDownClass` annotated with `@AfterAll`
 - This static method is run once after all the tests have been run and can be used to clean up external resources.
- `setUp` annotated with `@BeforeEach`
 - This nonstatic method is run once before each test method in the JUnit test class. It can be used to set things to a known good state before each test.
- `tearDown` annotated with `@AfterEach`
 - This nonstatic method is run once after each test method in the JUnit test completes. It can be used to clean up after each test.

We will use the `setUp` method to put our system into a known good state before each test. Setting the system to a known good state in `setUp` gives us everything we need right now.

To do this, we're going to add in a new `ClassRosterDao` property declaration to our test class and then use the `setUp` in the JUnit class to reference the new overloaded constructor we built in the `ClassRosterDaoFileImpl` with a newly created file to ensure that we aren't messing up production data and that our `testDAO` is effectively empty.

Update your `ClassRosterDaoFileImplTest` method, as shown in Listing 31.5, to declare our new `testDao` object, remove the unneeded setup/teardown methods and template method, and fill `setUp` with our initialization code.

LISTING 31.5

Updated `ClassRosterDaoFileImplTest` Method

```
public class ClassRosterDaoFileImplTest {  
    ClassRosterDao testDao;  
  
    public ClassRosterDaoFileImplTest() {  
    }  
  
    @BeforeEach  
    public void setUp() throws Exception{  
        String testFile = "testroster.txt";  
        // Create a new FileWriter for the test roster file  
        new FileWriter(testFile);  
        testDao = new ClassRosterDaoFileImpl(testFile);  
    }  
}
```

This way, before every test run, we will have created a new blank `testroster.txt` file using the `FileWriter` and then used that as our `fileName` when instantiating our `testDao`. Both ensure that we are starting with a fresh, empty DAO object and minimize our interference with the normal application's data stored in the `roster.txt` file.

Arrange/Act/Assert for Stateful Code

We must add two general steps to the **Arrange** part of the unit testing process for stateful code.

1. Put the system in a known good state using the JUnit `setUp` method.

2. Create test data and add it to the system before each test is run. This can be done either in the `setUp` method (if the test data should be the same for all tests) or in each individual test method.

The Act and Assert parts of the testing process are identical to those for testing stateless code:

- **Act:** Call the method under test with the correct test inputs
- **Assert:** Verify that the expected result matched the actual result

NOTE As you learned in the previous lesson, Arrange/Act/Assert is also referred to as Given/When/Then.

CLASS ROSTER DAO TEST DESIGN

Our `ClassRosterDao` has four methods that we must test.

- Add Student
- Get Student
- Get All Students
- Remove Student

When designing tests for these methods in our DAO, we need to make sure we design and implement tests for each method. In most cases, we will use more than one per test to “arrange” our DAO’s state correctly before we assert on it.

First, we must consider what the effects of each method will be on the DAO and then how we can use the other tools (DAO methods) available to us to determine whether the state was correctly changed. Let’s start this process with the `addStudent` method.

Add Student

Let’s look at the DAO interface method and comment, which is shown in Listing 31.6, before we get started on designing a test for this method.

LISTING 31.6 The DAO Interface Method and Comment

```
/**  
 * Adds the given Student to the roster and associates it with the given  
 * student id. If there is already a student associated with the given  
 * student id it will return that student object, otherwise it will  
 * return null.  
 * @param studentId id with which student is to be associated  
 * @param student student to be added to the roster  
 * @return the Student object previously associated with the given  
 * student id if it exists, null otherwise  
 */  
Student addStudent(String studentId, Student student)  
throws ClassRosterPersistenceException;
```

Reading through both the method and its JavaDoc comment, let's list some of the information we can gather about this method and its potential testing requirements. Here is what we know about `addStudent`:

- To call this method, we will need a `studentId` String and a `Student` object.
- After calling this method with those parameters, the student should now be stored within `ClassRosterDaoFileImplTest` and be associated with the given `studentId`.
- If there was no student in the DAO already with that `studentId`, the return should be null.
- However, if there was a student stored under that ID, then the previous `Student` associated will be returned.
- There is also a chance that this method could throw a `ClassRosterPersistenceException`, which implies it is touching a data store (we know it should be reading and writing to the file).

All this data is something we should already know, but it is always a good idea to double- and even triple-check your assumptions when beginning the testing process.

Looking at our gathered information, the first two bullet points are particularly important for our testing purposes. In this case, they focus on the fact that `addStudent` changes the state by adding a new or replacing an existing student within the DAO.

NOTE Remember when testing stateful code, you are generally most interested in arranging and asserting on how the code changes the state.

However, just the state change isn't the only thing we need to focus on. We need to also understand how to assert and prove that the state change happened correctly. In this case, we use other methods to measure the effect that this method call had on the state—the change of students stored within the DAO.

Remember, the ClassRosterDaoFileImplTest has three other methods: getStudent, getAllStudents, and removeStudent. Technically, each one of these could be used to verify the effect that calling the addStudent method has on the DAO, but getStudent is particularly well suited in that it allows us to simply retrieve the student from the DAO and examine it.

Get Student

Before we use it in a test, let's refresh ourselves with what getStudent does. Listing 31.7 presents the current code for getStudent.

LISTING 31.7

The getStudent Interface Method and Comments

```
/**  
 * Returns the student object associated with the given student id.  
 * Returns null if no such student exists  
 *  
 * @param studentId ID of the student to retrieve  
 * @return the Student object associated with the given student id,  
 * null if no such student exists  
 */  
Student getStudent(String studentId)  
throws ClassRosterPersistenceException ;
```

We know the following things about getStudent:

- To call this method, we will need a studentId String.
- The return should be a Student object. In theory, this is a student previously stored under that ID.

- If there was no student in the DAO with that studentId, the return should be null.
- There is also a chance that this method could throw a ClassRosterPersistenceException, which again implies touching a persistent data store.

ClassRosterDaoTest: Adding and Getting a Student

Our Add/Get Student test is straightforward. We know that the DAO is in an empty state since we created a new empty instance within our setUp method.

- The first step of this test is to create a new Student object (Arrange).
- Then we add that student to the DAO (Act).
- Next, we get the student back out of the DAO and put it in another variable (Act).
- Finally, we check to see that the data within the stored student is equal to the retrieved student from the DAO (Assert).

Listing 31.8 presents our test code.

LISTING 31.8

The testAddGetStudent Test Code

```
@Test
public void testAddGetStudent() throws Exception {
    // Create our method test inputs
    String studentId = "0001";
    Student student = new Student(studentId);
    student.setFirstName("Ada");
    student.setLastName("Lovelace");
    student.setCohort("Java-May-1845");

    // Add the student to the DAO
    testDao.addStudent(studentId, student);
    // Get the student from the DAO
    Student retrievedStudent = testDao.getStudent(studentId);

    // Check that the data is equal
    assertEquals(student.getStudentId(),
                retrievedStudent.getStudentId(),
                "Checking student id.");
    assertEquals(student.getFirstName(),
                retrievedStudent.getFirstName(),
                "Checking student first name.");
    assertEquals(student.getLastName(),
                retrievedStudent.getLastName(),
                "Checking student last name.");
}
```

```
        retrievedStudent.getLastName(),
        "Checking student last name.");
    assertEquals(student.getYear(),
        retrievedStudent.getYear(),
        "Checking student cohort.");
}
}
```

Our first test is now complete, but we've only touched two of our DAO methods, and we only used the test with one Student object. Let's do another.

Get All Students

Let's look at the DAO interface method and comment before we get started on designing a test for the getAllStudents method. This is shown in Listing 31.9.

LISTING 31.9

The getAllStudents Interface Method and Comments

```
/** 
 * Returns a List of all Students on the roster.
 *
 * @return Student List containing all students on the roster.
 */
List<Student> getAllStudents()
    throws ClassRosterPersistenceException;
```

This is an easier method to consider. It takes in no parameters and simply returns a list of students within the DAO. To properly test this method, we're going to have to use it alongside addStudent and maybe removeStudent to change what is contained within the list.

ClassRosterDaoTest: Adding and Getting All Students

This test is slightly more complicated than the previous Add/Get test, but not by much. Here we are focusing on testing the two methods addStudent and getAllStudents and verifying that each method works. To do this properly, we should really use multiple Student objects. In this test, we do the following:

1. Create and add two Student objects to the DAO (Arrange).
2. Get all the Student objects from the DAO (Act).
3. Check to see that the DAO returned the two objects (Assert).

Listing 31.10 shows the resulting test code.

LISTING 31.10

The testAddAndGetAllStudents Test Code

```
@Test
public void testAddAndGetAllStudents() throws Exception {
    // Create our first student
    Student firstStudent = new Student("0001");
    firstStudent.setFirstName("Ada");
    firstStudent.setLastName("Lovelace");
    firstStudent.setCohort("Java-May-1845");

    // Create our second student
    Student secondStudent = new Student("0002");
    secondStudent.setFirstName("Charles");
    secondStudent.setLastName("Babbage");
    secondStudent.setCohort(".NET-May-1845");

    // Add both our students to the DAO
    testDao.addStudent(firstStudent.getId(), firstStudent);
    testDao.addStudent(secondStudent.getId(), secondStudent);

    // Retrieve the list of all students within the DAO
    List<Student> allStudents = testDao.getAllStudents();

    // First check the general contents of the list
    assertNotNull(allStudents, "The list of students must not be null.");
    assertEquals(2, allStudents.size(), "The list of students should have 2
students.");

    // Then the specifics
    assertTrue(testDao.getAllStudents().contains(firstStudent),
        "The list of students should include Ada.");
    assertTrue(testDao.getAllStudents().contains(secondStudent),
        "The list of students should include Charles.");
}
```

IMPORTANT NOTE

The test in Listing 31.10 will fail unless you properly completed the equals/hashCode additions to your Student class discussed earlier in this lesson.

Remove Student

Let's look at the DAO interface method and comment for removing a student before we get started on designing a test for this method. Listing 31.11 shows the removeStudent interface method.

LISTING 31.11

The removeStudent Interface Method and Comments

```
/**  
 * Removes from the roster the student associated with the given id.  
 * Returns the student object that is being removed or null if  
 * there is no student associated with the given id  
 *  
 * @param studentId id of student to be removed  
 * @return Student object that was removed or null if no student  
 * was associated with the given student id  
 */  
Student removeStudent(String studentId);
```

We know the following things about removeStudent:

- To call this method, we will need a studentId String.
- The return should be a Student object. In theory, this is a student that was previously stored under that ID.
- If there was no student in the DAO with that studentId, the return should be null.
- The state effect of this on the DAO is that the returned student should no longer be within the DAO.
- There is another mention of a ClassRosterPersistenceException, which implies more persistent data storage.

ClassRosterDaoTest: Adding and Removing Students

In this test, we do the following:

1. Create and add two Student objects to the DAO (Arrange).
2. Remove one of the students from the DAO (Act).
3. Check to see that there is only one student left in the DAO (Assert).
4. Check to see that the DAO returns null if we try to retrieve the removed student (Assert).

5. Remove the other student from the DAO (Act).
6. Check to see that there are no students in the DAO (Assert).
7. Check to see that the DAO returns null if we try to retrieve the removed student (Assert).

Even though this is only one test, as shown in Listing 31.12, we are acting and asserting several times. This is perfectly fine. You can act and assert as many times as needed to fully execute the test case.

LISTING 31.12

The testRemoveStudent Test Code

```
@Test
public void testRemoveStudent() throws Exception {
    // Create two new students
    Student firstStudent = new Student("0001");
    firstStudent.setFirstName("Ada");
    firstStudent.setLastName("Lovelace");
    firstStudent.setCohort("Java-May-1945");

    Student secondStudent = new Student("0002");
    secondStudent.setFirstName("Charles");
    secondStudent.setLastName("Babbage");
    secondStudent.setCohort(".NET-May-1945");

    // Add both to the DAO
    testDao.addStudent(firstStudent.getId(), firstStudent);
    testDao.addStudent(secondStudent.getId(), secondStudent);

    // remove the first student - Ada
    Student removedStudent = testDao.removeStudent(firstStudent.
getStudentId());

    // Check that the correct object was removed.
    assertEquals(removedStudent, firstStudent, "The removed student should
be Ada.");

    // Get all the students
    List<Student> allStudents = testDao.getAllStudents();

    // First check the general contents of the list
    assertNotNull( allStudents, "All students list should be not null.");
```

```
        assertEquals( 1, allStudents.size(), "All students should only have 1 student.");

        // Then the specifics
        assertFalse( allStudents.contains(firstStudent),
                     "All students should NOT include Ada.");
        assertTrue( allStudents.contains(secondStudent),
                     "All students should NOT include Charles.");

        // Remove the second student
        removedStudent = testDao.removeStudent(secondStudent.getId());
        // Check that the correct object was removed.
        assertEquals( removedStudent, secondStudent,
                     "The removed student should be Charles.");

        // retrieve all of the students again and check the list.
        allStudents = testDao.getAllStudents();

        // Check the contents of the list - it should be empty
        assertTrue( allStudents.isEmpty(),
                     "The retrieved list of students should be empty.");

        // Try to get both students by their old id. They should be null.
        Student retrievedStudent = testDao.getStudent(firstStudent.getId());
        assertNull(retrievedStudent, "Ada was removed, should be null.");

        retrievedStudent = testDao.getStudent(secondStudent.getId());
        assertNull(retrievedStudent, "Charles was removed, should be null.");
    }
}
```

UNIT TESTING THE SERVICE LAYER

In this part of the lesson, we will unit test the service layer for the Class Roster application. As mentioned at the beginning of this lesson, we will use a new testing technique called *stubbing*.

planning the Test Design

The main purpose of the unit test suite for the service layer is to test that the business rules are being applied properly. For completeness, we will also test the pass-through methods of the service layer. We will test these business rules:

- Creation of a Student object with an existing student ID is prohibited.
- All fields on the Student object must have nonempty values.

These business rules are enforced in the `createStudent` method of the DAO. We will need three test cases:

- Create valid student. All fields have values, and the student does not have an existing student ID.
- Create a student with an existing student ID.
- Create a student with one or more empty field values.

We will also implement tests for `getAllStudents`, `getStudent`, and `removeStudent`. These tests will be similar to the tests we created for the matching DAO methods in a previous step.

Creating the Test Class

You should remember this step from our `DAOImpl` testing step. We will create a JUnit test class that will hold our test suite for our service layer. Do the following:

1. In the NetBeans Project view, right-click the `ServiceLayerImpl` class.
2. Select *Tools*, and then select *Create/Update Tests*.
3. Walk through the wizard to create a new JUnit class.
 - The name should look similar to `classroster.service.ClassRosterServiceLayerImplTest`.
 - Ensure that *Test Packages* is selected in the *Location* drop-down.
 - Generated Code Checkbox List should have everything selected.
 - Method Access Levels Checkbox List should have nothing selected.
 - Generated Comments Checkbox List should have nothing selected.
4. Once these steps are finished, click the **OK** button.

Completing the wizard should leave you with a new Java test class, as shown in Listing 31.13, where we will write our service layer test suite.

LISTING 31.13

ClassRosterServiceLayerImplTest

```
public class ClassRosterServiceLayerImplTest {  
    public ClassRosterServiceLayerImplTest() {  
    }  
  
    @BeforeAll  
    public static void setUpClass() {  
    }  
  
    @AfterAll  
    public static void tearDownClass() {  
    }  
  
    @BeforeEach  
    public void setUp() {  
    }  
  
    @AfterEach  
    public void tearDown() {  
    }  
  
    @Test  
    public void testSomeMethod() {  
        fail("The test case is a prototype.");  
    }  
}
```

Creating the DAO Stubs

The service layer is not responsible for storing or retrieving Student objects. That is the job of the DAO. This means we don't have to (or want to) test the actual persistence of Student objects in this test suite. We only want to test the business rules and the integration between the service layer and the DAO.

Given this fact, we can use stubbed versions of our DAOs to test the functionality of the service layer. Since we are programming to interfaces and using constructor-based dependency injection, it will be easy to use the stubbed versions of the DAOs instead of the file-based implementations.

A **stubbed version** of a component simply returns canned data for any method call. We can set up a stubbed version of a component to act just about any way we want or need it to. Let's take a look at the stubbed implementation of our DAOs.

ClassRosterAuditDaoStubImpl

The **ClassRosterAuditDaoStubImpl** implementation shown in Listing 31.14 is simple. The `writeAuditEntry` method does nothing. This allows the service layer to make the call to the Audit DAO, but nothing will get written to the audit log file.

This is a testing class and belongs inside our test packages, not our development sources. Therefore, to create this testing stub, proceed with the following steps:

1. In the NetBeans Project view, right-click the `ClassRosterServiceLayerImplTest`'s test package.
2. Select **New** and then **Java Class**.
3. Name it **ClassRosterAuditDaoStubImpl**.
4. Click the *Finish* button.
5. Update the internal code to the shell implementation in Listing 31.14.

LISTING 31.14

The **ClassRosterAuditDaoStubImpl** Shell Implementation

```
public class ClassRosterAuditDaoStubImpl implements ClassRosterAuditDao {  
  
    @Override  
    public void writeAuditEntry(String entry) throws  
        ClassRosterPersistenceException {  
        //do nothing . . .  
    }  
}
```

ClassRosterDaoStubImpl

This **ClassRosterDaoStubImpl** implementation is a bit more complicated than the Audit DAO stub implementation, but it is still quite straightforward. The stub implementation has the following features:

- A member field of type `Student`

This represents the one and only student in the DAOStub.

- **Constructor**
We have two. One is a no-arg constructor that instantiates a hard-coded student for our stub. The other allows a test student to be injected via the constructor by a test class.
- **addStudent**
This returns our `onlyStudent` field if the ID matches our `onlyStudent`'s ID. Otherwise, it returns null. Note that there is no persistence. The incoming `Student` parameter is never added to the `DAOStub` or persisted in any way.
- **getAllStudents**
This method simply returns a List containing the one and only student.
- **getStudent**
This returns our `onlyStudent` field if the ID matches our `onlyStudent`'s ID; otherwise, it returns null.
- **removeStudent**
This returns our `onlyStudent` field if the ID matches our `onlyStudent`'s ID; otherwise, it returns null. Note that this does not change or remove our `onlyStudent` existence within our `DAOStub`.

To create this stub, follow similar steps as the `AuditDaoStub` previously, and then fill out your `ClassRosterDaoStubImpl` with the code in Listing 31.15.

LISTING 31.15

ClassRosterDaoStubImpl

```
public class ClassRosterDaoStubImpl implements ClassRosterDao {
    public Student onlyStudent;

    public ClassRosterDaoStubImpl() {
        onlyStudent = new Student("0001");
        onlyStudent.setFirstName("Ada");
        onlyStudent.setLastName("Lovelace");
        onlyStudent.setCohort("Java-May-1845");
    }

    public ClassRosterDaoStubImpl(Student testStudent){
        this.onlyStudent = testStudent;
    }
}
```

```

@Override
public Student addStudent(String studentId, Student student)
    throws ClassRosterPersistenceException {
    if (studentId.equals(onlyStudent.getStudentId())) {
        return onlyStudent;
    } else {
        return null;
    }
}

@Override
public List<Student> getAllStudents()
    throws ClassRosterPersistenceException {
    List<Student> studentList = new ArrayList<>();
    studentList.add(onlyStudent);
    return studentList;
}

@Override
public Student getStudent(String studentId)
    throws ClassRosterPersistenceException {
    if (studentId.equals(onlyStudent.getStudentId())) {
        return onlyStudent;
    } else {
        return null;
    }
}

@Override
public Student removeStudent(String studentId)
    throws ClassRosterPersistenceException {
    if (studentId.equals(onlyStudent.getStudentId())) {
        return onlyStudent;
    } else {
        return null;
    }
}
}

```

Test Setup

By using stubbed DAOs for these tests, we have essentially defined the state of our test system. Since the DAOs are the properties that belong to our service layer, their state represents the state of our service layer. That means if we can start these DAOs in a good

state, so will our service layer. Since our stubs start in a predefined state, we will begin testing our service layer with the assumption that it contains a Class Roster DAO with exactly one student.

Although we don't have to do any work to put our code in a known good state, we do have to create the service layer object and wire in our stub DAOs. This code is similar to the code we have in the App class of the Class Roster application. Update the top of your ClassRosterServiceLayerImplTest with the setup in Listing 31.16.

LISTING 31.16

Create the Service Layer Object and Wire a Stub DAO

```
private ClassRosterServiceLayer service;

public ClassRosterServiceLayerImplTest() {
    ClassRosterDao dao = new ClassRosterDaoStubImpl();
    ClassRosterAuditDao auditDao = new ClassRosterAuditDaoStubImpl();

    service = new ClassRosterServiceLayerImpl(dao, auditDao);
}
```

Test Implementation

Finally, we will look at our test implementations. These include testCreateValidStudent, testCreateStudentDuplicateId, testGetAllStudents, testGetStudent, and testRemoveStudent.

testCreateValidStudent

The testCreateValidStudent test shown in Listing 31.17 is quite straightforward. We are simply asserting that the creation of a valid student (no duplicate student ID and values for all fields that the only hard-coded student in our DaoStub) does not cause an exception to be thrown.

LISTING 31.17

The testCreateValidStudent test

```
@Test
public void testCreateValidStudent() {
```

```

// ARRANGE
Student student = new Student("0002");
student.setFirstName("Charles");
student.setLastName("Babbage");
student.setCohort(".NET-May-1845");
// ACT
try {
    service.createStudent(student);
} catch (ClassRosterDuplicateIdException
        | ClassRosterDataValidationException
        | ClassRosterPersistenceException e) {
// ASSERT
    fail("Student was valid. No exception should have been thrown.");
}

```

testCreateStudentDuplicateId

The `testCreateStudentDuplicateId` test shown in Listing 31.18 asserts that a `ClassRosterDuplicateIdException` is thrown when trying to create a student with an existing student ID. In this case, we know that the stubbed implementation of the Class Roster DAO has an existing student with an ID of 0001 so we attempt to create a new student with that student ID. Since we expect an exception to be thrown by this call, we surround it with a `try/catch`.

- If the call executes and no exception is thrown, we fail the test with a message saying we expected an exception to be thrown.
- If the expected exception is thrown, we simply return. Since there are no errors or exceptions, this lets the JUnit test framework know that this test passed.
- If a different exception is thrown, the test will fail.

LISTING 31.18

The `testCreateStudentDuplicateId` Test

```

@Test
public void testCreateDuplicateIdStudent() {
    // ARRANGE
    Student student = new Student("0001");
    student.setFirstName("Charles");
    student.setLastName("Babbage");
    student.setCohort(".NET-May-1845");
}

```

```

// ACT
try {
    service.createStudent(student);
    fail("Expected DupId Exception was not thrown.");
} catch (ClassRosterDataValidationException
        | ClassRosterPersistenceException e) {
// ASSERT
    fail("Incorrect exception was thrown.");
} catch (ClassRosterDuplicateIdException e){
    return;
}
}

```

testCreateStudentInvalidData

The `testCreateStudentInvalidData` test shown in Listing 31.19 is similar to the test for duplicate student ID. Here we ensure that we don't have a duplicate student ID and then also leave one of the fields blank. We use the same `try/catch` techniques as shown in the previous example except we are looking for a different exception.

LISTING 31.19

The `testCreateStudentInvalidData` Test

```

@Test
public void testCreateStudentInvalidData() throws Exception {
    // ARRANGE
    Student student = new Student("0002");
    student.setFirstName("");
    student.setLastName("Babbage");
    student.setCohort(".NET-May-1845");

    // ACT
    try {
        service.createStudent(student);
        fail("Expected ValidationException was not thrown.");
    } catch (ClassRosterDuplicateIdException
            | ClassRosterPersistenceException e) {
// ASSERT
        fail("Incorrect exception was thrown.");
    } catch (ClassRosterDataValidationException e){
        return;
    }
}

```

test GetAllStudents

In the test GetAllStudents test shown in Listing 31.20, since we know that the stubbed Class Roster DAO contains only one student, we assert that only one student is returned from the getAllStudents service layer method.

LISTING 31.20

The test GetAllStudents Test

```
@Test
public void test GetAllStudents() throws Exception {
    // ARRANGE
    Student testClone = new Student("0001");
    testClone.setFirstName("Ada");
    testClone.setLastName("Lovelace");
    testClone.setCohort("Java-May-1845");

    // ACT & ASSERT
    assertEquals( 1, service.getAllStudents().size(),
                  "Should only have one student.");
    assertTrue( service.getAllStudents().contains(testClone),
                "The one student should be Ada.");
}
```

test GetStudent

For the test GetStudent test shown in Listing 31.21, since we know that the stubbed Class Roster DAO only contains one Student with Student ID = 0001, we assert that a Student is returned when we ask for Student ID 0001 and that no student is returned when we ask for Student ID 0042.

LISTING 31.21

The test GetStudent Test

```
@Test
public void test GetStudent() throws Exception {
    // ARRANGE
    Student testClone = new Student("0001");
    testClone.setFirstName("Ada");
    testClone.setLastName("Lovelace");
    testClone.setCohort("Java-May-1845");
```

```

// ACT & ASSERT
Student shouldBeAda = service.getStudent("0001");
assertNotNull(shouldBeAda, "Getting 0001 should be not null.");
assertEquals( testClone, shouldBeAda,
    "Student stored under 0001 should be Ada.");

Student shouldBeNull = service.getStudent("0042");
assertNull( shouldBeNull, "Getting 0042 should be null.");
}


```

testRemoveStudent

The behavior of the `removeStudent` method is that it will remove the student and return the associated `Student` object if a student exists for the given student ID. Otherwise, it will do nothing and return null. In the `testRemoveStudent` test shown in Listing 31.22, we assert that a `Student` object is returned when we remove student ID 0001 and that null is returned when we remove student ID 0042.

LISTING 31.22

The `testRemoveStudent` Test

```

@Test
public void testRemoveStudent() throws Exception {
    // ARRANGE
    Student testClone = new Student("0001");
    testClone.setFirstName("Ada");
    testClone.setLastName("Lovelace");
    testClone.setCohort("Java-May-1845");

    // ACT & ASSERT
    Student shouldBeAda = service.removeStudent("0001");
    assertNotNull( shouldBeAda, "Removing 0001 should be not null.");
    assertEquals( testClone, shouldBeAda, "Student removed from 0001 should
be Ada.");

    Student shouldBeNull = service.removeStudent("0042");
    assertNull( shouldBeNull, "Removing 0042 should be null.");
}

```

SUMMARY

In this lesson, we discussed and demonstrated the techniques for unit testing stateful code. We then completed the unit tests for the Class Roster service layer and saw how stubbed implementations can be used to test other components. The following are the main points to remember from this lesson:

- You must put stateful code into a known good state before unit testing it.
- You must be mindful of the order in which you test and call methods for stateful code. Calls to one method can affect the results from other methods.
- JUnit provides set up and tear down methods to help put stateful code into a known good state.
- Implementing the equals and hashCode methods on your DTOs can make unit testing a lot easier. Remember that you must use the same fields to calculate equality and the hash code for your objects.
- Stubbed implementations of components such as DAOs can be used to test components that use them.
- Hard-coded stubbed component implementations have a fixed state.
- When testing components (such as the service layer) that depend on other components (such as DAOs), you must wire the dependencies together as part of your overall test setup.
- The combination of a try/catch block and the JUnit static method called fail can be used to test conditions where an exception is expected.

EXERCISES

Many people learn best by doing, so this section includes exercises using what you learned in this lesson and previously in this course.

Exercise 1: Testing the Address Book app

Exercise 2: Testing the DVD Library program

Exercise 1: Testing the Address Book App

Design and implement a complete set of unit tests for the DAO of the Address Book application that you created in Lesson 27. Use the lesson notes as a guide and pattern your test suite after the test suite created in the Class Roster unit test code-along earlier in this lesson.

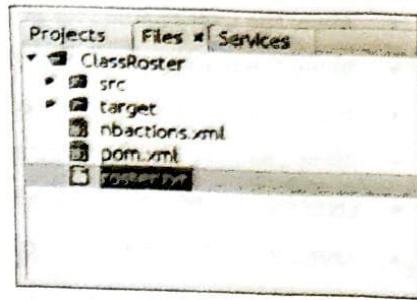


Figure 27.12 The `roster.txt` file has been added

SUMMARY

You have created your first MVC CRUD application. Remember to follow the steps we took to build this application as you build the exercises included with this lesson. Make sure you use the code you wrote here as a template, or guide, for your future projects.

EXERCISES

These exercises help you practice in your own application what you have reviewed in this lesson. These are to do on your own.

Exercise 1: Your Own DVD Library

Exercise 2: Electronic Address Book

NOTE This exercise will be referenced again in a future lesson where you will extend what you build here.

Exercise 1: DVD Library Update

The purpose of this exercise is to demonstrate your proficiency in what you've learned up to this point, including basic Java syntax involving console input and output, basic file input and output, basic string/text manipulation, flow of control statements, expressions, and basic data structures such as arrays, lists, and maps. Additionally, you will demonstrate your proficiency in implementing the MVC design pattern and dependency injection.

In this exercise, you will create a program that stores information about a DVD collection. The program must do the following:

- Allow the user to add a DVD to the collection
- Allow the user to remove a DVD from the collection

- Allow the user to edit the information for an existing DVD in the collection
- Allow the user to list the DVDs in the collection
- Allow the user to display the information for a particular DVD
- Allow the user to search for a DVD by title
- Load the DVD library from a file
- Save the DVD library back to the file when the program completes
- Allow the user to add, edit, or delete many DVDs in one session

Additionally, you should follow the MVC design pattern and follow the agile approach checklist outlined in Appendix D.

Your DVD data transfer object should have the following fields:

- Title
- Release date
- MPAA rating
- Director's name
- Studio
- User rating or note (allows the user to enter additional information, e.g., "Good family movie")

Exercise 2: Electronic Address Book

The objective of this exercise is to continue to practice designing programs that consist of more than one class, implement the MVC pattern presented in this course, and use Java Collections and Maps. In this exercise, design a program that acts as an electronic address book. This program should do the following:

- Allow the user to add addresses to the address book
- Allow the user to remove addresses from the address book
- Allow the user to see how many addresses are in the book
- Allow the user to list all the addresses in the book
- Allow the user to find an address by last name

Design an Address class to hold address information. Design an AddressBookDao class that holds Address objects. It should also have methods to do the following:

- Add an address
- Remove an address

- Find an address by last name
- Return a count of addresses in AddressBookDao
- Return all the addresses in AddressBookDao

Design an AddressBookController class. This class should do the following:

- Orchestrate all activities of the program.
- Use the view and UserIO class from a previous exercise to handle all console input and output. You might need to modify the view.
- Use the AddressBookDao class to store Address objects.

You should also add code to allow the user to edit an address and to initialize your address book from a file. Finally, add code to allow the user to save an address back to the file.

The following provides ideas as a sample for the UI for the application:

=====

Initial Menu:

Please select the operation you wish to perform:

1. Add Address
2. Delete Address
3. Find Address
4. List Address Count
5. List All Addresses

Add Address Menu:

Please Enter First Name:

Please Enter Last Name:

Please Enter Street Address:

.

.

.

Address added. Press 1 to go to Main Menu.

Delete Address Menu:

Please enter last name of address to delete:

John Doe

123 Main Street

Hometown, OH, 12345

Really Delete?

Address Deleted. Press 1 to go to Main Menu.

Exercise 2: Testing the DVD Library

GRAMMAR

Design and implement a complete set of unit tests for the DAO of the DVD Library application that you created in Lesson 27. Use the information from this lesson as a guide and pattern your test suite after the test suite created in the Class Roster application.

EXERCISES

Many people feel that grammar is the most important part of English. In this lesson, we will focus on some common mistakes that can occur when writing in English. We will also look at how to correct these mistakes.

Exercise 3: Testing the Application

Design and implement a complete set of unit tests for the DAO of the DVD Library application that you created in Lesson 27. Use the information from this lesson as a guide and pattern your test suite after the test suite created in the Class Roster application.

SUMMARY

Once you have made the changes described in this lesson, you will be able to run your unit tests and your application just as you did before. We made no changes to functionality; we simply refactored the project by replacing the code we wrote that manually did dependency injection and replaced it with the Spring dependency injection container. Use this project as a template for future Maven/Spring projects.

Here are the important points to remember:

- We have to add dependency entries for all external libraries to the Maven POM file. Specifically, for this project, we added entries for Spring Core.
- The Spring container (known as the *application context*) is configured via an XML file that contains definitions for all the objects we want Spring to control.
- Maven makes a distinction between the application and test environments. We must have a Spring configuration file for each environment.
- We place the Spring configuration files on the classpath. In our Maven projects, we place these files in either *Other Sources* or *Other Test Sources*.
- To use the Spring container, we must instantiate an implementation of `ApplicationContext` (in our case, we use `ClassPathXmlApplicationContext`). The `ApplicationContext` will read the Spring configuration file and instantiate all configured objects. We then ask the `ApplicationContext` for references to these objects by name rather than instantiating them ourselves.

EXERCISES

This exercise helps you practice what you are learning about using the Spring framework. This is to do on your own.

Exercise 1: DVD Library

Exercise 1: DVD Library

Convert your DVD Library application to use the Spring DI container. Use what you learned in this lesson as a guide and pattern for your approach.

PART VI

Appendices

Appendix A: Code Checklist

Appendix B: Java Concepts Study List

Appendix C: Agile Approach Checklist for Console CRUD Applications

Appendix A: Code Checklist

This document can be used as a guide to help you prepare your code. Many organizations have similar lists with more detail to guide you through preparing your code before sharing with a team. Go through the items on this checklist before you consider your code to be complete.

FUNCTIONALITY

- Make sure your code compiles.
- Double-check that the application performs per specification.
- Have unit tests to verify that the application performs per specifications.
 - Use positive unit tests to exercise code as intended.
 - Use negative unit tests to intentionally misuse the code and verify robust error handling and bounds of input values.
 - Step through all possible code paths in the debugger.
- Confirm that code analysis tools in your IDE do not report warnings.

STYLE

- Use a consistent style.
 - Ensure that classes, methods, and variables are named with appropriate letter casing and code blocks are indented properly.
- Follow the Single Responsibility Principle.
 - Strive to separate large functions into collections of smaller functions that perform single tasks.
 - Strive to make classes cohesive so that each class has one area of responsibility that it handles completely.

- Use comments appropriately.
 - Use comments sparingly; intent should be clear from your naming conventions.
 - Use comments to clarify your intent if it might not be clear, for example, when there is a legitimately complicated section of code to satisfy a business rule or requirement.
 - Do not use comments to make up for poorly written code; they should enhance already readable code.
 - Do not comment the obvious.

FUNCTIONALITY

- Write tests about code's outcome.
- Don't check pre-existing assumptions by description.
- Have unit tests to verify that the software behaves the way it's intended.
- Use descriptive unit tests to express what the code does.
- Use JUnit and JUnit assertions.
- Setup properly so dependencies can be injected.
- Consider what code suggests (oops! I wrote `doTest()` to test `main()`)

STYLE

- Use a consistent style.
- Ensure class, method, and variable names reflect their purpose.
- Group and code blocks into reasonable blocks.
- Follow the single responsibility principle.
- Stick to interfaces instead of concrete implementations unless forced to.
- Form logical passes.
- Separate interface from implementation.

- What kinds of variables are created in memory?
- What is garbage collection?
- How does the garbage collector know what memory to release?
- Is the Java language pass by value or pass by reference?

Appendix B: Java Concepts Study List

The following is a list of basic Java concepts that you learned within this book. You should know and be comfortable discussing these topics in preparation for technical interviews and your first days on the job.

VARIABLES

- What is a variable?
- What is a variable declaration?
- What is a variable assignment?
- What does the `final` keyword mean?
- What does it mean to cast a variable?

METHODS

- What makes up a method's signature?
- What is a method parameter?
- What is a method body?
- What does the `static` keyword mean?
- What does the `final` keyword mean?
- What does the term `override` mean?
- What does the term `overload` mean?

ARRAYS

- What is an array?
- What does an array hold?
- What are the limitations of an array?
- Is an array an object?
- The first value in an array can be found at what index?
- What is an `ArrayIndexOutOfBoundsException`?

MAKING DECISIONS

- What is a Boolean expression?
- What is a switch statement?
- What is an if statement?
- What is an else statement?
- What is an else if statement?

LOOPS

- What is a loop?
- What is a for loop?
- What is a for-each loop?
- What can a for loop do that a for-each loop cannot?
- What is a while loop?
- What is a do-while loop?
- What is the difference between a while loop and a do/while loop?

JAVA/JVM

- How is the import keyword used?
- What is a package?
- What is Javadoc?
- What is the stack?

- What kinds of variables are created there?
- What is garbage collection?
- How does the garbage collector know when to return memory to the heap?
- Is the Java language pass by value or pass by reference?
- What does pass by value mean?
- What does pass by reference mean?
- What is a NullPointerException?

OBJECTS

- What is a class?
- What is an object?
- What is the heap?
- What is scope?
- What is an accessor?
- What is a mutator?
- What is a constructor?
- What does the public keyword mean?
- What does the static keyword mean?
- What does the private keyword mean?
- What does the abstract keyword mean?
- How is abstract used in conjunction with classes?
- What is polymorphism?
- What does encapsulation mean?
- What does it mean if a class is cohesive?
- What is the single responsibility principle?
- What is data hiding?
- What does the term generic mean?
- What are the two categories of nested classes?
- What is a local class?
- What is a static nested class?

- What level of access to the properties and methods of the enclosing class does a static nested class have?
- Under what conditions can a local class access variables in the enclosing scope?
- What is the default constructor?

INTERFACES

- What is an interface?
- How do interfaces help achieve polymorphism?

INHERITANCE

- What is inheritance?
- What does it mean if one class is derived from another class?
- What does it mean if one class extends another class?
- How does inheritance help achieve polymorphism?
- What is a base class?
- What is an abstract base class?
- What is a superclass?
- What is a subclass?
- What is specialization?

N-TIER DESIGN

- What does MVC stand for? What is it?
- What is a model?
- What is a controller?
- What is a view?
- What is a DAO?
- What does it mean to be loosely coupled?
- Why is it a good idea to split your applications into layers?

COLLECTIONS

- What is an ArrayList?
- What is a List? How is it different from an ArrayList?
- What is a HashMap?
- What is a Map? How is it different from a HashMap?
- What is a Collection?
- What is an iterator?

EXCEPTIONS

- What is an exception?
- What is an unchecked exception?
- What is a checked exception?
- What is a runtime exception?
- What does a finally block do?
- What is a try/catch?
- What does the throws keyword mean?

LAMBDAS/STREAMS

- What is a lambda?
- What is a stream (in the context of the new Java 8 Collection API)?
- What are aggregate operations?
- What is a pipeline?
- What is a functional interface?

SPRING CORE

- What are the four design principles of Spring?
- What is dependency injection?
- What is constructor injection?
- What is setter injection?

Appendix C: Agile Approach Checklist for Console CRUD Applications

This appendix outlines an approach that can be used to break down a project and implement it in a stepwise and agile way. This will allow you to deliver demonstrable, testable code in small increments.

NOTE A hypothetical address book application is used as an example when describing this approach.

ASSUMPTIONS

Our applications are mainly concerned with creating, reading, updating, and deleting (CRUD) data to and from persistent storage (files or databases).

- Our programs (even the simple ones) must be split into tiers or layers as practice for more complicated problems. This leads us to create projects with the following features:
 - Data transfer objects (DTOs), also known as domain objects (e.g., Address)
 - A data access object that handles storing data in memory and reading and writing data from persistent storage (e.g., AddressBookDao)
 - Some kind of user interface (e.g., the console)
 - Logic that orchestrates the objects in the program (this will be found in the controller)

REQUIREMENT STEPS

Step 1

Create user stories from the problem statement and/or requirements. For Address Book, they would be as follows:

- Add an address
- View an address
- Delete an address
- Edit an address
- List all addresses
- Show the number of addresses in the address book
- Save to file
- Read from file

DESIGN STEPS

Step 2

Analyze the problem statement and identify the required classes. For Address Book, we have the following:

- Address (DTO)
- AddressBookDao (data access object)
- UserIO (helper used by the View class to interact with the console)
- View class (used by Controller to handle user interaction)
- AddressBookController (this class orchestrates the program)
- App (this class has a main method that instantiates AddressBookController and calls the execute method)

Step 3

Flesh out the classes by defining properties and methods for each.

Step 4

Create a flowchart for the user interaction process (e.g., displaying menus and reacting to user menu choices).

Step 5

Create a file format to match the identified domain object(s) (e.g., Address). For example, for storing addresses, we might have something like this:

```
<firstName>::<lastName>::<streetAddress>::<city>::<state>::<zip>
```

CONSTRUCTION STEPS

Step 6

Create a menu system (in the execute method of AddressBookController) with stubbed-out code for each menu choice.

For example, when the user presses the choice to add an address, the system simply prints a message saying, "Add Address: To Be Implemented." This can be delivered to the user for testing and feedback.

Step 7

Pick a user story to implement, for example "Add address." For this story, we'll need to do the following:

1. Create the Address class (domain object).
2. Create the AddressBookDao class.
 - a. Include a HashMap or ArrayList to hold the Address objects as a class-level variable.
 - b. Implement the addAddress(...) method.
3. Add code into the AddressBookController to do the following:
 - a. Instantiate AddressBookDao object for storing Address information.
 - b. Read in address information from the user.
 - c. Create a new Address object.

- d. Put address information from the user into the Address object.
- e. Add the new Address object to the AddressBookDao.

Each user story will be different, and the first one is always the most work. Repeat step 7 for each user story.

rectangle 270
single-dimensional 271
dimensional 271
elements 273, 275
accepts multiple input stages
jagged arrays/multidimensional arrays
multidimensional arrays
array 270–271 being
produced by arrays
single-dimensional
array 272–273
arrays 272–273
summing up 272, 273
error handling 273–274, 275–276
index 273–274 array 273–274
initializing 277–278, 279–280, 281–282
jagged 278–279, 280–282
rectangle 280–281, 282–283
single-dimensional 283–284
iteration
priming condition 284–285
jagged array 284–285, 286–287
multidimensional
array 284–285, 286–287
looping through 285–286
two-dimensional arrays 285–286
multidimensional 285–286
multiple 285–286, 287–288, 289
multidimensional 287–288
one-dimensional 287–288, 289–290
single 291–292
bottom C 292
salary 292
end 292
single-dimensional 292–293
array 292–293
multiple 293–294
one-dimensional 293–294
array 293–294
autumn 294