0101 - Decisions (Chapter 5)

The content in this chapter is in progress.

It is incomplete and likely full of typos. In each section, I’ve referenced the corresponding section in the required textbook (see Canvas). If you don’t want to deal with any of that, you can choose to just read Chapter 5 in that book.

***Note: the videos in this chapter go a little beyond the content in the corresponding sections—​they show how to write the code, but also show how to apply some things to our class and object design. If you skip the videos, you might miss information needed for the assignments and/or quiz.***

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

[CIS150AB Course Materials Feedback (Google Form)](https://forms.gle/4173pZ1yPuNX7pku6)

*Source code examples from this chapter and associated videos are available on*[*GitHub*](https://github.com/timmcmichael/EMCCTimFiles/tree/4bf0da6df6f4fe3e3a0ccd477b4455df400cffb6/OOP%20with%20Java%20(CIS150AB)/05%20Decisions)*.*

Boolean Expressions

Until now, our programs have all executed very sequentially and predictably—​one line of code after another. How boring! To give our programs the ability to *branch* and execute different code based on different conditions, we need to introduce the concept of *decisions*.

In computer programming, a decision is made based on a **Boolean expression**, which is an expression that evaluates to either true or false. Think of them as questions that can be answered with a simple "yes" or "no". *Is this student’s GPA high enough to qualify for the scholarship? Has this cell phone customer used all of their data? Did the user enter "exit"?*

The true or false value of a Boolean expression can be used to determine which code block will execute next, so it’s important to understand how they work.

Time To Watch!

Until now, our programs have all executed sequentially.

[Start animation]

When the first line of code is finished, we move to the second; when that's done, we go on to the third line. When there are no more lines to execute, the program stops.

And every time we run the program, it's going to flow through the same sequence of steps in the same order.

In order to give our programs some flexibility in how they execute, we need to introduce the concept of decisions, which allow the program to run blocks of code based on certain conditions.

[Advance to conditional statements]

To make decisions, our programs will use Boolean logic. Boolean logic is all based on the idea of true or false, which means that our program can make decisions based on yes or no questions only--there's no "maybe" in Boolean logic.

Is the amount the user entered greater than 10? That's either true or false. If it's true, we can execute a block of code. [CLICK]

If it's not true, we'll skip that block and move on with the program.

[CLICK]

At the heart of Boolean logic are Boolean variables and Boolean expressions.

[Next slide]

A Boolean variable can only have one of two values: true or false. In Java, we use the data type boolean to store these values.

[Click to show code]

`boolean` variables are generally named with a question or a statement that can be answered with true or false. `hasCar` is true or false, right? The valid values you can put in a boolean variable are `true` or `false`, in all lowercase and no quotes.

We'll see uses for `boolean` variables as we go along.

[Next slide]

The other common structure in Java that can be used for decisions is a Boolean expression. A Boolean expression is a statement that evaluates to true or false. Like arithmetic expressions, Boolean expressions use binary operators--operators that work on two operands. The most common boolean operations are comparisons, and you'll be familiar with these from your math classes.

The addition operation takes two operands to find a sum.

[Next slide]

"Relational operators" also take two operands, but they evaluate to a Boolean value. For example, 15 > 25 evaluates to false--15 is not greater than 25.

[Next slide]

There are six relational operators in Java. For our examples, we'll assume we have a couple of int variables defined.

[Chart enter]

The first operator is the equality operator, `==`.

[Next highlight]

It checks if two values are equal. For example, age is equal to 17, which evaluates to false. This operator leads to a really common beginner mistake: using `=` instead of `==`.

[Next - arrow]

A single = symbol *\*always\** assigns a value to a variable, and a double == symbol *\*always\** checks for equality. When I speak code out loud, I alway pronounce == as "is equal to" instead of "equals" to make it clear which one I mean.

[Next highlight]

The next operator is the not equal operator, `!=`. It checks if two values are not equal. For example, age is not equal to 17, which evaluates to true.

[Next highlight]

Less than and greater than work as you'd expect. These examples show that the two operands can be variables, literal values, or a combination of both.

And the last two operators are less than or equal to and greater than or equal to. Again, they work as you'd expect--but notice the symbols we use. In math, we got used to drawing specific symbols for these operations

[Next animation - symbols]

But we don't have characters like that on our keyboard, so we type them the way we'd speak them: "less than or equal to" is a less than symbol and the equals symbol. "Greater than or equal to" is a greater than symbol and the equals symbol.

[Next slide]

There are a few more operators we use in boolean logic, which we call "logical operators." These operators allow us to combine multiple boolean expressions into a single, more complex expression. We'll look at these closer when we start writing decision-making code, but for now, here they are:

[Next - drop table]

Two ampersands, or what you probably think of as two "and" symbols, is the logical AND operator.

[Next highlight]

It returns true only if both operands are true. This example evaluates to true if age is greater than or equal to 18 AND age is less than or equal to 21. If age is too low or too high, the expression evaluates to false. It basically checks to see if age is between 18 and 21.

[Next - highlight]

The operator for "OR" is two vertical lines. These characters are called "pipe lines", or just "pipes". They're just above the return or enter key on your keyboard.

Logical OR returns true if at least one of the operands is true. This example evaluates to true if age is less than 18 OR the isRegistered variable is false. This expression checks to see if the user is either too young or not registered to vote.

[Next - highlight]

Last up is the logical NOT operator, which is an exclamation point. It inverts the value of a boolean expression. If the expression is true, NOT makes it false. If the expression is false, NOT makes it true.

[Next - remove highlight]

There's actually another logical operator called "exclusive or, but we don't use that and we'll ignore it for now since this chart is probably confusing enough already. As I said, we'll make sense out of these once we're able to start using them in our code.

[Next Slide - strings]

We often want to test the equality of strings, like when we ask the user to enter a password.

[Print variable assignment]

We'll need to check if what they entered matches the password we have stored.

If these variables were numbers, we could just use the == operator

[Next == ]

But in Java, we can't use == to compare strings. Actually, we *\_can\_* use == to compare strings, but it doesn't work the way you'd expect--and it's not going to give you the results you expect. In short, don't do it.

[Next - red line]

Instead, we have to use a String method called `equals`, and we pass in the value we want to compare to.

[Next - .equals]

Incidentally, it doesn't matter which string use use to call the `equals` method and which we compare to. The result will be the same, so you could also write `password.equals(userPassword)`.

[Next - last slide]

It might not be clear yet why we're spending time on this, but understanding Boolean logic is going to make it possible to write code that makes decisions. And that's what we'll look at next...

Boolean Values

The simplest way to use Boolean in Java is with the keywords true and false. These can be assigned to variables of type boolean:

boolean isTimAmazing = true;

boolean isClassBoring = false;

Relational Operations

Also known as *comparison operations*, relational operations are expressions that compare two values. You will remember them from math class. *Is*x*greater than*y*? Is*a*less than or equal to*10*? Is*c*equal to*d*?* Relational operators, like arithmetic operators, are *binary* operators that require two operands. In other words, you need two values to compare. In your math class, you could draw symbols that aren’t on your keyboard, like a ≥ for "greater than or equal to." In Java, where we have to stick with stuff on our keyboard, we use the following symbols:

|  |  |
| --- | --- |
| *Table 1. Java relational operators* | |
| **Operator** | **Description** |
| == | Equality (checks if two values are equal) |
| != | Inequality (checks if two values are not equal) |
| < | Less-than |
| > | Greater-than |
| <= | Less-than-or-equal-to |
| >= | Greater-than-or-equal-to operator. |

A relational operation will always evaluate to either true or false. We can store that result in a boolean variable, as shown below, and we’ll also learn how to use these expressions in if statements later in this chapter.

boolean canBuyAlcohol = age >= 21;

boolean isNegative = number < 0;

These assignment statements work like any other: the expression to the right of the equals symbol is evaluated, and the result is stored in the variable on the left.

Testing a String for Equality

In Java, you can’t use the == operator to compare two strings. The following code will not work as you might expect:

*Example of an incorrect*String*comparison.*

// Should output "true", but (sometimes) doesn't

String name = "Paul McCartney";

System.out.print("Is this person Paul McCartney?");

System.out.println(name == "Paul McCartney"); // DON'T DO THIS!

The == operator does not compare the contents of the strings, but rather the memory addresses where the strings are stored. Due to the nuances of how the Java runtime handles String objects, this code will work sometimes, by coincidence, but it’s not reliable.

Instead, you’ll need to use the equals() method, which is a method of the String class and gets called using dot notation. This method will examine the contents of the strings and return true if they are the same, and false if they are different.

*Example of a correct*String*comparison.*

// Outputs "true"

String name = "Paul McCartney";

System.out.print("Is this person Paul McCartney? ");

System.out.println(name.equals("Paul McCartney"));

if Statements

The if statement is the most basic decision-making structure in Java. It allows you to execute a block of code only if a certain condition is true.

The syntax of an if statement is the keyword if, followed by a Boolean expression in parentheses, followed by a block of code in curly braces. If the Boolean expression evaluates to true, the block of code will execute. If the Boolean expression evaluates to false, the block of code will be skipped. In either case the program will continue executing the next line of code after the if-block.

*Example of an*if*Statement.*

int age = 20;

if (age < 21) {

System.out.println("You can't buy alcohol.");

}

System.out.println("Keep that in mind when you go to the store!");

In this example, the if statement checks if the variable age is less than 21. Since 20 is less than 21, the Boolean expression evaluates to true, and the block of code inside the if statement is executed—​and it prints "You can’t buy alcohol." The program then continues to the next line of code, which prints "Keep that in mind when you go to the store!"

If the value of age were 22, the Boolean expression would evaluate to false, and the block of code inside the if statement would be skipped. The program would then continue to the next line of code, which prints "Keep that in mind when you go to the store!"

Keep in mind, the parentheses after the if keyword can contain any Boolean expression—​not just this simple example.

Adding an else Block

An if statement simply determines whether or not to execute a single block of code. If you want to choose between two blocks of code, you can add an else block to the if statement. The syntax is simple: after the if block, add the keyword else, followed by a block of code in curly braces.

*Example of an*if-else*Statement.*

int age = 20;

if (age < 21) {

System.out.println("You can't buy alcohol.");

}

else {

System.out.println("You can buy alcohol.");

}

System.out.println("Keep that in mind when you go to the store!");

An if-else statement will always execute one block of code or the other, but never both. Basically, it’s an either-or situation.

Time To Watch!

In this video, we will take a look at if and if-else statements. These conditional statements give us the ability to run different blocks of code based on a Boolean espression. If you're not familiar with Boolean expressions, be sure to watch the previous video first.

In order to jump right in to these examples, we're stepping away from our OOP perspective for a moment and just working within a main() method. But don't worry, we'll be back to OOP in one of the next videos on this topic.

If you know me, you know that I really do love my teaching job. But I also love money, and the rest of my family loves fancy name brand clothes--and that's a problem. To solve that problem, I think I'll start selling some of my hilarious jokes. I'm going to write a program that calculates a total purchase price based on the number of jokes the customer wants to buy. I'll also give a discount if the customer buys 10 or more jokes. Let's get started.

Jokes sell for $30 each, which sounds expensive--but I know my worth. If a customer buys 10 or more jokes, they get a 10% discount. Through the magic of video editing, I'll add a header with test cases and I'll get the Scanner stuff ready to go.

It's easy to forget those test cases, so keep building the habit of adding them right at the start. They are worth points, and if you consistently leave them out--our magician friend can predict your future.

[You shall not pass]

Test cases help us understand the code we need to write, starting with the variables we'll need. In this program I have one input and one output--so I need at least two variables. That might not be the only variables I need, but I can create those two start.

I like to initialize local variables when I declare them. With numeric types, I tend to initialize them to negative one. This is really just a reminder--if I see a negative number when I run my program, I know I forgot to assign a value. A lot of people use zero, and that's fine, too.

I'll prompt the user for the number of jokes they want to buy, and I'll store that in the numJokes variable I just declared. This is a good place to stop and test the program so far.

Obviously it doesn't do a lot yet, but at least I know it's working so far.

With that working, I'll calculate a total price by multplying the number of jokes by the price per joke.

And I'll go ahead and output that total price.

And I'll run the program and try my first test case.

It looks good. Notice that the price only "kinda" looks like a currency value. As beginners, we don't worry about making the output look pretty--we just want the program to work correctly.

To make sure it is working correctly, I'll run this a few more times with my other test cases.

They all work until I get to 10 or more jokes, because obviously I haven't written the code to a discount yet. But because I did those test cases, I have already worked out how the math works. If they buy enough for the 10% discount, it means they're really just paying 90% of the total. 90% as a decimal is 0.9, so I'll multiply the total price by 0.9 to get the discounted price.

And for now I'll just add that code and try it out.

The program works with those higher numbers, but now it doesn't work if they're buying just a few jokes.

We need the program to make a decision about whether to not to apply the discount, and we'll do that with an if statement.

Start with the keyword `if`, followed by a set of parentheses. Inside the parentheses, we'll put a boolean expression that determines whether the discount should be applied. They have to buy 10 or more, so I'll start with this boolean express:

if (numJokes > 10)

and a code block. I'll move my discount calculation into that code block so that it only runs if the boolean expression is true. And let's test that.

We're really close, but when they buy exactly 10 jokes, they don't get the discount. This is where it's good to have thoughtful test cases. Calculation changes when they buy ten jokes, so we call that an edge case. It's important to test edge cases to make sure our decision logic is correct.

I've made a really common mistake. My boolean expression is `numJokes > 10`, but I need to use `numJokes >= 10` to include the edge case. I could also write it `numJokes > 9`, but I think the greater-than-or-equal-to syntax is more clear.

Let's test that again.

Our if statement is good now.

Okay, if I'm going to sell enough jokes to pay for all that lululumon, I need to entice my cheap customers to buy a little more and get that discount. So I think we'll add a message for customers who don't buy enough jokes to get the discount.

This means we're going to have two different blocks of code, and we only want one or the other to run. We can do that by adding an `else` block to our `if` statement.

The `else` keyword immediately follows the `if` code block, and then we add another code block.

Now we have these two code blocks. If the boolean expression is true, the first block runs. If it's false, the second block runs. I'll add my enticing message to the else block so people who don't get the discount know what they're missing.

Let's take a look and try a couple different values.

Well, it looks like my money problems are solved, so I'm going to wrap up this program and start shopping for vacation homes. We'll look at some other things we can do with if-else statements in the the next video.

Interesting!

Java includes a shorthand form of an if-else statement called the *ternary operator*, which uses the question mark symbol. It’s a useful little trick, but it can be confusing for beginner—​and for the people reading your code later. We won’t look at them in this course, but a web search should turn up plenty of examples if you are curious.

Assignments in my course require you to actually use if-else statements, so you shouldn’t use the ternary operator in code you submit to me.

The if-else if Structure

The video for this section is currently in progress and will be available soon.

The if-else structure is great for choosing between two blocks of code, but what if you have more than two options? To handle this, you can chain multiple if-else statements together.

WeatherRecord.java*(excerpt). Example of an*if-else if*statement. See*[*GitHub*](https://github.com/timmcmichael/EMCCTimFiles/tree/4bf0da6df6f4fe3e3a0ccd477b4455df400cffb6/OOP%20with%20Java%20(CIS150AB)/05%20Decisions)*for the full file.*

public class WeatherRecord {

// Fields

private String date;

private int highTemperature;

private double averageWindSpeed;

public String getTempDescription() {

if (this.highTemperature > 90) {

return "Hot";

} else if (this.highTemperature > 70) {

return "Warm";

} else if (this.highTemperature > 50) {

return "Cool";

} else {

return "Cold";

}

}

}

In this example, the getDescription() method will return a String that describes the weather based on the high temperature of the day.

* If the high temperature is greater than 90, the method will return "Hot".
* If the high temperature is greater than 70, the method will return "Warm".
* If the high temperature is greater than 50, the method will return "Cool".
* If the high temperature is 50 or less, the method will return "Cold".

The code begins with the first statement, and if it evaluates to true, the corresponding block of code will execute. If the first statement evaluates to false, the program will move on to the next else if statement, and so on. Once a code block is executed, it will hit a return statement, which will exit the method and not evaluate any other blocks of code. Therefore, only one block of code will execute. If the program gets through the entire structure without finding a true condition, it will execute the block of code in the else block, if one is present.

If you’re looking at that code critically, you might notice that the else block is not strictly necessary. You could just put the return "Cold"; statement at the end of the method, and it would work the same way. However, that depends on the logic of the if-else if structure and whether or not you’re using return statements in the blocks of code.

Interesting!

The above example shows one clever use of decisions in an object class. You’d think that a getDescription() method would return the value of a field called description, but there is no field. Instead, it just uses a decision structure to return a description based on the highTemperature field.

In summary, An if-else if structure can execute, at most, one block of code. If an else block is included at the end, it guarantees that exactly one block of code will execute.

Time To Watch!

It might seem like if-else statements are pretty limited. After all, the only thing they can do is pick between two possible options. But we're going to learn a few tricks to make them more powerful. One of those tricks is the *\*if-else if\** statement. An *\*if-else if\** statement is a way to choose one code block from a bunch of options.

We want to start integrating decisions into our OOP code, so for this video we're going to work with a class called K12Student, with a demo program called K12StudentDemo. The code for both files is available from the GitHub repository for this course, which is linked in description and in the eBook.

This is going to be a class that just keeps track of a student's age and grade level.

[Show diagram of K12Student class]

There's no constructor yet, and we'll save that until close to the end. There are a couple other inconsistencies here: we have a getSchoolType() method, which should return "elementary school," "middle school," or "high school", but we don't have a schoolType field to go with it. And with gradeLevel(), we have kind of the opposite--we're missing a set method for that field. These are all intentional and we'll talk about them as we go.

But before we get to that, we're going to apply what we already know about if statements to our understanding of encapsulation. Remember that encapsulation is the idea that we should keep our fields private and only allow access to them through methods.

Our getters have been fine, since all they are supposed to do is return the current value of the field. Until now, though, our setters have just assigned whatever value they receive to the field. That's not very effective encapsulation, since it still allows other code to mangle our data. But with if statements, we can make our setters a little smarter and a little more secure.

Let's start with the gradeLevel field. The grade level is determined the student's age, so we don't want a situation where those two don't match. If the student is 16, we don't want some other code to come along and set the grade level to 3. So we're not going to have a setter for gradeLevel--we'll just set that value in our setAge() method.

Speaking of setAge(), we're allowing the age to be set to any int value, but if we're talking about a K-12 student, we know that the age should be between about 5 or 6 years old and about 18. If the value passed in is outside that range, we'll need to address it.

Let's just add an if statement below that assignment to see if our age is below 6--and to keep things simple, we'll say that the student needs to be 6 or older.

[this.age < 6]

If that boolean expression is true, our age is too low, so we'll just set it to 6. There are other ways we could handle this--maybe we just leave the age what it was before, which is a common approach, or maybe we set it to some kind of flag value that indicates an error, like -1. When we learn more about programming, we might trigger something called an exception. But as beginners, we'll just set it to 6.

And now after that if block, I'll add a similar one to check if the age is above 18. If it is, we'll set it to 18.

[add that code]

Notice that our code is setting the value to the parameter passed in, then is making an adjustment if necessary. We could also write decision statements before the assignment, if that's what we prefer.

Let's test this out by trying to set the age out of range and seeing what happens.

Now, before we move on from this, we mentioned that we can use the age to determine the grave level--and that's why we don't have a setGradeLevel() method. Let's just set the grade level to the age minus 6. I know that in reality it's not quite that simple, but it demonstrates the point that our setter can be a little more complex than just assigning a value.

And we'll test that out as well by outputting that getGradeLevel() method.

That's obviously not a perfect system, but it's an improvement over just having a bunch of setters that don't check anything.

And now we can move on to the getSchoolType() method.

There's no field for schoolType, because that is completely tied to the grade level. For our purposes, students attend elementary school until 5th grade, then they go to middle school for 6th through 8th, and then high school for 9 through 12.

Let's add an if statement to check if the grade level is less than 6. We could also say less than or equal to 5, it really amounts to the same thing in this case. If that expression is true, we'll return "elementary school". Again, no need for a field to store that, we'll just figure it out any time this getter gets called. The code has to run each time, but the benefit is that we don't have to worry about keeping the field in sync with the grade level.

And we'll test that out to see how we're doing so far.

Okay, with that part working, we can now add our else statement. But here's where we have a problem. An if-else structure lets us choose between two options, but we have three options here. If I put "middle school" here, then I don't have a way to address the high school students.

So what we can do is just tack another if statement onto the else keyword, and this will check if the gradeLevel is less than 9. This is called an *\*if-else if\** statement. If the first condition is false, it will check this one. If this one is false, it will check the else block, if there is one.

So in this body we'll return "middle school."

And if the student isn't in elementary or middle school, they must be in high school, so now we'll just have an else block that returns "high school". This makes "high school" the default, essentially. If for some reason a student had a gradeLevel of 0, they'd be in high school. So that's not great, but it reminds us why encapsulation is so important.

Let's run our demo program again to see how we did.

Alright, with the tricky part of the code working, the last thing I want to look at is the constructor. Let's create a constructor with parameters for the name and age. Remember, as long as we have the age, we can figure out the grade level, and as long as we have the grade level, we can figure out the school type--so these two are the only pieces we need.

Name is no big deal, since it doesn't have any restrictions--we we'll just assign that. Our age does have restrictions though, so we need to make sure it's not out of range. It's tempting to just copy and paste that code from our setter because I don't want to type it all out again. But any time we find ourselves copying and pasting code, we should ask if there's a better way. In this case, we can just call the setAge() method, which will take care of all that for us. And it means that if we ever change the way we handle age, we only have to change it in one place.

So instead of an assignment statement, we'll just call this.setAge() and pass in the are parameter.

We've broken our demo class--remember that the compiler won't make a default constructor any more since I've defined a constructor of my own. So we'll just go over to that file and comment out the default constructor calls, and then replace them with instantiations that use our new constructor.

And we'll go ahead and run that.ß

Okay, so this class is now wrapped up. We've seen how we can use decisions to improve our encapsulation, and we've also seen how we can use if-else if statements to choose from more than two options. I've still got a couple decision tricks left, but we'll keep going in the next video.

*Note: This video also shows how to use*if*statements in an OOP context, specifically to help with encapsulation.*

Nested if-else Statements

The video for this section is currently in progress and will be available soon. In the meantime, you can read section 5.4 (starting page 172) in the required textbook.

If we want a block of code to execute only if two different conditions are met, we can place if statements inside of each other—​which is called *nesting*. Nested if statements check multiple conditions in a hierarchical way: if one condition is met, it will proceed and check the next condition; if the first condition is not met, it will skip the inner if block.

*Basic structure of a nested*if-else*statement.*

if (condition1) {

if (condition2) {

// executes if both condition1 and condition2 are true

}

else {

// executes if condition1 is true and condition2 is false

}

}

else {

// executes if condition1 is false

}

In the example below, the outermost if-else structure checks the high temperature of the day. The if-else structures within those blocks checks the average wind speed and returns an appropriate description.

WeatherRecord.java*(excerpt). Example of a nested*if-else*statement. See*[*GitHub*](https://github.com/timmcmichael/EMCCTimFiles/tree/4bf0da6df6f4fe3e3a0ccd477b4455df400cffb6/OOP%20with%20Java%20(CIS150AB)/05%20Decisions)*for the full file.*

public String getFullDescription() {

if (this.highTemperature > 90) {

if (this.averageWindSpeed > 10) {

return "Hot and Windy";

} else {

return "Hot";

}

} else if (this.highTemperature > 70) {

if (this.averageWindSpeed > 10) {

return "Warm and Windy";

} else {

return "Warm";

}

} else if (this.highTemperature > 50) {

if (this.averageWindSpeed > 10) {

return "Cool and Windy";

} else {

return "Cool";

}

} else {

if (this.averageWindSpeed > 10) {

return "Cold and Windy";

} else {

return "Cold";

}

}

}

Time To Watch!

We've learned a couple ways to use if statements to add some flexibility to our decisions. Sometimes we need to make what we might call a "hierarchical" decision, where we make one big decision, and once we've done that we drill down into a more specific choice.

For example, when the server hands you a menu at a restaurant, you first have to decide if you're going to order from the breakfast, lunch, or dinner options. Once you've made that decision, you can look at the specific choices for that meal. In coding, we can make this kind of hierarchical decision with a nested if statement.

I'm an old fashioned guy. Well, I'm an old guy, but I'm also old fashioned. When I was young and there was no Spotify, we'd go to the record store to get music we wanted. Like everyone else my age, I always thought it'd be cool to own a record store. So we're going to consider a simple record store inventory system to learn about nested if statements.

We're going to work with a class called `VinylRecord`, with a demo program called `VinylRecordDemo`. The code for both files is available from the GitHub repository for this course, which is linked in description and in the eBook.

This is going to be a class for an inventory and pricing application for a record store. The shop prices albums based on their genre.

[Show price chart]

This chart shows a few genres and their corresponding prices. General "Rock" albums are $15, but "Classic Rock" and "Grunge" cost a little more, at 20 and 25 dollars. And while general Blues albums are $25, Chicago Blues and Delta Blues are priced higher.

And anything that's not a Rock or Blues album gets a price of $10.

Just like our restaurant menu example, determining the price of a specific record is a hierarchical decision. We first need to decide if the album is Rock, Blues, or something else. Once we've made that decision, we can make a more specific decision to establish the price.

Here's a diagram of the class we'll use. First, notice that there are a couple methods on the diagram that aren't in our code: we're missing the constructor and the `setGenre()` method.

And there are a couple other things to point out. The `setGenre()` method doesn't follow our usual pattern--it's one setter for two different fields. And there's no `setPrice()` method at all, so there's no way to get a value into that field.

I've designed it that way because, since the price is determined by the genre and subgenre, there's no need to set the price directly. In fact, if we let someone set the price directly they might price it wrong. So we'll take care of setting the price in the `setGenre()` method.

Okay, let's go ahead and create the header for `setGenre()`. It's a void method with parameters for the genre and subgenre. The body will be like other setters, but with two assignments statements instead of one.

Once we've set those fields, we can set the price. First up, we'll create an if block to check if the genre is "Rock". Remember that we can't use the `==` operator to compare strings; we'll use the `equals()` method.

If it is a Rock album, we still need to check the subgenre, so for now, I'll just set the price to $15, which is the default for a rock album, and get the rest of this "top level" decision in place.

If it's not a Rock album, we need to check if it's a Blues album. That means adding an else-if statement with another equals() call. The default for a Blues album is $25, so we'll set the price to that.

And finally, we need a catch-all else block for anything that's not Rock or Blues. We'll set the price to $10 in this block.

If we think of that idea of a hierachical decision, this would be the top-level decision--it's the decision we have to make first, like breakfast, lunch, or dinner.

This is a good place to go over to the demo class and make sure it's working so far.

// Create a three records, Rock, Blues, and Hip-Hop.

This shows that our top-level decision structure is working, so now we can focus on the next level of decisions.

Inside the block for our Rock albums, we need to look at the subgenre of the record. My if statement will check to see if it's Classic Rock; if so, we'll set the price to $20.0.

We'll need an else-if statement to check for Grunge albums, and set the price to $25.0.

And we'll need an else block to catch any other Rock albums, which is that $15.0 default.

Let's give that a shot and see how we're doing.

Okay, so now it's just a matter of applying the same kind of structure to our Blues genre. The first if block will check for Chicago Blues, and set the price to $30.0.

The else-if block will check for Delta Blues, and set the price to $35.0.

And the else block will set the price to $25.0.

All of this is followed by our else statement that sets the price to $10.0, but if we had more genres, we could just keep adding else-if blocks and then if-else statements within those for subgenre pricing.

We should be ready for our final test now.

And that's how we can use nested if statements. This gives us a way to make broad decisions first, and then drill down into more specific decisions. This is a common pattern in programming, and it's a good way to structure our code when we have a series of related decisions to make.

*Note: This video also shows how to use*if*statements in an OOP context, specifically to help with encapsulation.*

Using Logical Operators

In addition to the relational operators, Java also includes logical operators we can use to make more complex Boolean expressions. A logical operator is a binary operation, so it takes two operands—​but the operands are Boolean expressions instead of numbers.

|  |  |  |
| --- | --- | --- |
| *Table 2. Java logical operators* | | |
| **Operator** | **Name** | **Description** |
| && | AND | Evaluates to true if **both** operands are true |
| || | OR | Evaluates to true if **either** operand is true |
| ! | NOT | Evaluates to true if the operand is false; evaluates to false if the operand is true |

These operators can be used to combine multiple Boolean expressions into a single, more complex expression. For example, you could check if a student is eligible for a scholarship based on both their GPA (3.5 or better) and their age (younger than 25).

*Example of a decision using a logical AND operation.*

if (gpa >= 3.5 && age < 25) {

System.out.println("You qualify for the scholarship!");

}

In this example, the && operator is used to combine two Boolean expressions. The if statement will only execute the block of code if both expressions are true.

|  |  |
| --- | --- |
|  | Often, the logic you create using an AND operation can be implemented using nested if-else statements, and vice versa. |

The OR operation is similar, but only one of the expressions needs to be true for the entire expression to be true.

*Example of a decision using a logical OR operation.*

if (isTimAmazing || isJavaFun) {

System.out.println("You should take this class!");

}

Both operands in an AND or OR operation have to be complete Boolean expressions. Put another way, each side of the && or || operator must be able to evaluate to true or false on its own. The following code is a very common beginner mistake and will **not** compile:

if (percentage >= 80 && < 90) { …​ }

This reads like "if the percentage is greater than or equal to 80 and less than 90," but the second part of the expression is not a complete Boolean expression. You need to include the variable name and the comparison operator on both sides of the && operator.

The NOT operation is a little different, as it only takes one operand (making it a *unary* operator\_, if you’ve nerdy about words, like I am). It simply inverts the value of the operand. If the operand is true, the NOT operation will evaluate to false. If the operand is false, the NOT operation will evaluate to true.

*Example of a decision using a logical NOT operation.*

if (!isTimAmazing) {

System.out.println("At least his mom still loves him!");

}

Range Checking

There are a lot of situations where you might need to combine multiple conditions to make a decision, but one of the most common is *range checking*. Range checking means you want to see if a value is within a certain range.

A common example of range checking is to convert a percentage grade to a letter grade.

*Example of range checking using logical operators.*

public String getLetterGrade(int percentage) {

if (percentage >= 90 && percentage <= 100) {

return "A";

} else if (percentage >= 80 && percentage < 90) {

return "B";

} else if (percentage >= 70 && percentage < 80) {

return "C";

} else if (percentage >= 60 && percentage < 70) {

return "D";

} else if (percentage >= 0 && percentage < 60) {

return "F";

} else {

return "Invalid percentage";

}

}

The AND operator && used in this example means that in order to return "B", for example, the percentage must be greater than or equal to 80 *and* less than 90. If either of those conditions is not met, the program will move on to the next else if statement.

switch Statements

Java includes a structure called a switch statement that can be used to choose between multiple options. It is essentially another way to write an if-else if structure, but it can be more readable and easier to write in some situations. I generally consider switch structures to be optional—​you can complete all of the assignments in this course without using them—​but they are a useful tool to have in your programming toolbox. And since you see them often in code written by others, it’s good to know how they work.

The basic structure of a switch statement is as follows:

switch (expression) {

case value1:

// Code to be executed if expression equals value1

break;

case value2:

// Code to be executed if expression equals value2

break;

case value3:

// Code to be executed if expression equals value3

break;

default:

// Code to be executed if expression doesn't match any case

}

The expression in the parentheses after the switch keyword is evaluated, and then the program will jump to the case that matches the value of the expression. If there is no match, the program will execute the default block, if it is present.

The break statement is used to exit the switch block due to a behavior of switch that can be confusing to beginners, known as *fall-through*. If you don’t include a break statement at the end of a case block, the program will continue executing the code in the next case block, even if the value of the expression doesn’t match the case. This can be useful in some situations, but it’s generally not what you want, so you’ll usually see a break statement at the end of each case block.

*Example of a*switch*statement.*

public void trafficInstructions(String lightColor) {

switch (lightColor) {

case "red":

System.out.println("Stop!");

break;

case "yellow":

System.out.println("Slow down!");

break;

case "green":

System.out.println("Go!");

break;

default:

System.out.println("Invalid light color.");

}

}