

Chapter 3 - Decision Structures

Lecture notes on Ch 3 from Professor Arsenault

Gaddis Ch 3 Slides (3.1 - 3.4, 3.5 -3.10)

- Control flow (if)
 - if
 - if-else
 - nested if
 - if-else-if
 - Logical Operators (&& and ||)
 - Boolean Flags
 - Unary Operator (!)
 - Ternary Operator
 - Switch (classic and modern)
- Comparing String Objects
- Formatted output
 - printf
 - String format

If statements

If statements use **Relational Operators** to evaluate **Boolean Expressions**, which control the flow of a program.

What's a Relational Operator?

Relational operators (slide 7) are one type of [Operators in Java](#) (alongside Arithmetic, Assignment, Unary, Logical, and Ternary Operators, among others). These are the less than (<), greater than (>), equal (==) and not equal (!=) operators.

What's a Boolean Expression?

Remember the `boolean` primitive data type from Chapter 2? The `true` or `false` data type. `if` statements evaluate **Boolean Expressions** to decide if a section of code executes.

Boolean Expressions use Relational Operators to evaluate whether a statement is true or false:

- is `4 < 3` ? **true**
- is `25 > 5` ? **false**

Program executes is determined by the result of Boolean Expressions.

Simply put:

```
// If expression is true
if (true) {
    // do something
}
// If expression is not true
else {
    // do something else
}
```

Both conditions cannot both occur at the same time.

Here's a better example:

```
public class IfSamples {
    public static void main(String[] args) {
        double score = 55.5;

        if (score >= 59.9){
            System.out.println("You passed the class!");
        } else {
            System.out.println("See ya in the Spring.");
        }
    }
}
```

In the above, you either pass the class you or don't. The console output is determined by your score. Both statements cannot print on one run of the program!

Programming style of If statements

If there's only a single instruction after an `if`, curly brackets are not required. This is valid:

```
if (average > 95)
    grade = 'A';
```

This is also valid. You can write the statement on a single line:

```
if (average > 95) grade = 'A';
```

However, I write every `if` (and `else`) statement with curly braces, like this:

```
if (average > 95) {
    grade = 'A';
}
```

I highly recommend you use curly brackets every time for three reasons:

1. Curly braces help you to understand scope (what code belongs to which codeblock).
2. If you have more than one statement within the `if` block, you **must** use curly braces to enclose them (compiler error).
3. Curly braces help with code-readability by having a consistent alignment. They make nested `if` statements easier to read.

Speaking of curly braces... I prefer to use [Egyptian Brackets](#) (also called K&R style), which are seen in the above example. This is the style used by the Brian Kernighan and Dennis Ritchie in their legendary textbook [The C Programming Language](#) in 1978.

The alternative is where each brace gets its own line. Read more about [Notable Indentation Styles](#). See the Allman (also called BSD) Style.

Boolean variables as Flag

Using Boolean variables as flags (Slide 14) is a commonly-used and important programming practice that allows you to check the current state of an operation.

Here's an example. After we determine a score is above 59.9 (passing), we reassign a `passingScore` variable from `false` to `true`:

```
double score = 95.5;
boolean passingScore = false;

if (score > 59.9) {
    System.out.println("You passed the class!");
    passingScore = true;
}
```

```

}
else {
    System.out.println("See ya in the Spring.");
}

if (passingScore) {
    System.out.println("Prof Matra wants to offer you an internship.");
}

```

if statements only run if the expression is `true` (otherwise, only the `else` block is executed... if there is an `else` block).

In the above, if the expression `score > 59.9` does not evaluate to `true`, then `passingScore` does not change from `false` to `true`. After the first `if/else` statement executes, the program moves onto the second one. If `passingScore` is false, the block does not execute. But if `passingScore` is true, the block executes.

One great example would be with user login. Imagine you had an app that required a user to login:

1. Your app could have a `userLoggedIn=false` Boolean variable. User is not logged in yet.
2. When the user opens the app, there could be some limited options. She can see a bit of data, but not all, and the user might get a prompt to log in (think: viewing a social media app when NOT logged in).
3. A user clicks Login, you prompt the user for username and password (see: String Comparison methods). If it matches, set `userLoggedIn=true`.
4. Back on the home screen, new options could be shown to the user, such as their username, and additional pages such as "Edit profile". These additional options only display because the Boolean flag `userLoggedIn=true`

Comparing Chars

Remember how Char is a primitive type?

- This means that its value is stored directly in memory.
- Since we're talking about memory (low-level computer stuff), that value is not stored as 'A' or 'B' but as Hexadecimal (Unicode) numbers that have an order.
- Remember the table on Gaddis 2B slide 9:
 - 0x41 = A
 - 0x42 = B
 - 0x43 = C

- 0x50 = P
- 0x5A = Z
- 0x61 = a
- 0x70 = p
- 0x7A = z
- Is 'B' > 'A' ?
- Is 'A' < 'a' ?
- Upper case is stored at a LOWER number than lower case.

Nested If

Nested Ifs aren't the greatest programming practice, because there are better ways to reach the same solution that are less verbose.

They are good for complex decision making, for example, if you need to meet multiple conditions. See `LoanQualifier.java` for a good Nested If example.

You can start to avoid Nested Ifs by using Logical Operators (&&, ||).

Here's an example of nested ifs:

```
if (age > 18) {
    if (hasLicense) {
        System.out.println("You can drive.");
    } else {
        System.out.println("You need a license to drive.");
    }
} else {
    System.out.println("You are too young to drive.");
}
```

If we use a logical operator, we write less code, but we lose fidelity in our answer (we go from three possibilities to two)

```
if (age > 18 && hasLicense) {
    System.out.println("You can drive.");
}
else {
    System.out.println("You can't drive.");
}
```

One way to retain number of answers and write cleaner code is to incorporate logical operators and else if

```
if (age > 18 && hasLicense) {
    System.out.println("You can drive.");
} else if (age > 18) {
    System.out.println("You need a license to drive.");
} else {
    System.out.println("You are too young to drive.");
}
```

else if

What's the difference between many if statements, and an if-else-if statement?

- **Evaluation Flow:**
 - In multiple `if` statements, all conditions are evaluated unless explicitly terminated.
 - In `if-else if`, the evaluation stops at the first true condition.
- **Use Case:**
 - Use multiple `if` statements when each condition should be checked independently.
 - Use `if-else if` when only one condition should trigger an action, and further conditions should not be checked if one is already true.

If you want independent checks, use multiple `if` statements. If you want mutually exclusive conditions, use `if-else if-else`.

The best example of `else if` is `TestResults.java` because it removed choices categorically.

- It starts by checking if `testScore` is `<60`.
 - If it's not, the only thing we know is that `testScore` is `>59`.
- Then it checks if `testScore` is `<70`. By the point, we are testing whether `testScore` is between 60 and 69 (which corresponds to our grade of D).

Whenever it finds the correct range of the score, it prints the grade and stops execution.

Exercise: A good example with `TestResults.java` is to change every `else if` to `if` and run the program with different values. See how the output changes.

Relationship between Nested If, If-else-if, and Switch

Nested If

- **Use Case:** Useful when you need to check multiple conditions that depend on each other.
 - Loan: Must meet Salary req, and Years at Job req.
 - Can instead of logical operators (salary && yearsAtJob)

If-else-if

- **Use Case:** Useful for checking multiple conditions where each condition is mutually exclusive.

Switch

- **Use Case:** Ideal for checking a variable against multiple values. It is more readable and efficient than multiple `if-else-if` statements when dealing with many conditions.

Logical operators

- The **Unary** (YOU-nary) logical NOT operator (`!`) uses **one operand** to negate a Boolean expression.
- **Binary logical operator** (`&&` , `||`) combine if conditions using **two operands**. They can help you replace Nested If statements.
- **Ternary Conditional operator** takes **three operands** to function:
 - a condition,
 - a value to return if the condition is true,
 - and a value to return if the condition is false;

Unary operator

The unary NOT operator (`!`) in Java is used to invert the logical state of its operand.

The unary NOT operator is often used in `if` statements to execute code when a condition is false.

The unary NOT operator is a way to avoid writing an `else` statement by inverting the condition in an `if` statement. This is particularly useful when you only want to execute a block of code if a condition is false.

```
boolean isAuthenticated = true
if (!isAuthenticated) {
    System.out.println("User is not authenticated. Please log in.");
    //Prompt user to login or kick them out of the program!
}
// No need for an ELSE statement here. We are all good!
```

In this example, the message is printed only if `isAuthenticated` is `false`.

If `isAuthenticated` is `true`, the code inside the `if` block is skipped, and the method continues without needing an `else` statement.

Ternary Operator

Ternary Operator is called that because "ternary" means "consisting of three parts" in mathematical terms. It is also called the Conditional Operator.

The Ternary operator is a way to write the following if statement in a single line:

```
int x = 15;
int y = 5;

// Determine condition using If statement syntax
int z;
if (x > y)
    z = 10;
else
    z = 5;

// Determine condition using Ternary Operator
int z = (x > y) ? 10 : 5;
```

In `ConsultantCharges.java`, You enter the number of hours worked.

If you ever dealt with a consultant or a lawyer, they may have a minimum number of hours they can be paid for. In this case, if they work 4 hours, you have to pay them for 5. How could we write this using an if else statement?

Use cases for ternary operators

Ternary operators are sometimes seen as confusing, but they are useful:

1. **Simple Conditional Assignments:** Ternary operators are great for assigning values to variables based on a condition in a concise way. For example:

```
int num = 10;
String result = (num % 2 == 0) ? "Even" : "Odd";
```

2. **Inline Conditional Logic:** They can be used to simplify if-else statements that are short and straightforward, making the code more readable:

```
int a = 5, b = 10;
int max = (a > b) ? a : b;
```

3. **Default Values:** Ternary operators are often used to assign default values when a variable might be null:

```
String user = "Peter";
String name = (user != null) ? user : "Guest";
```

4. **Conditional Return Values:** They can be used to return values from functions based on conditions:

```
public String getStatus(int score) {
    return (score >= 50) ? "Pass" : "Fail";
}
```

5. **Conditional Printing:** They can be used to print different messages based on conditions:



```
System.out.println((age >= 18) ? "Adult" : "Minor");
```

String Comparison methods

Project 2B Tip: With String methods, you can combine multiple methods! Evaluated left to right. Example:

```
String menu = "Max's Fried Fish Dish"
String newMenu = menu.replace("Fish", "Chicken")
    .replace("Dish", "Dinner").toUpperCase();
```

Last Thursday night, Java made me look like an absolute fool. I told the class, *Hey class, you can't compare Strings like you do with primitives!* and then I proceeded to compare Strings like I did with primitives. I wrote up a complicated reason about it but [this article](#) does a better job at explaining it.

The bottom line: You compare primitives using equals logical operator (`5 == 4+1` ) but you shouldn't do this with Strings (`"Max" == "Max"` ), although sometimes it works. Strings have two comparison methods

String equals()

To compare the contents of two strings, always use the `.equals()` method:

```
String str1 = "Max";
String str2 = "Max";

if (str1.equals(str2)) {
    System.out.println("True");
} else {
    System.out.println("False");
}
// True
```

String equalsIgnoreCase()

If you want to test String equality but don't care about case differences (for example, in a username where capital letters don't matter):

```
String str1 = "Max";
String str2 = "max";

if (str1.equalsIgnoreCase(str2)) {
    System.out.println("True");
} else {
    System.out.println("False");
}
// True
```

Switch Statements

Java Switch syntax has changed over time! The `yield` keyword and the **arrow syntax** were introduced to enhance the functionality and readability of switch statements in Java. The following shows the modern vs. the classic way of writing Switch statements. Use the modern syntax!! 🛠️

Modern switch example

Here's the Switch example we did in class:

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter your grade: ");
String gradeString = keyboard.nextLine();
char gradeLetter = gradeString.toUpperCase().charAt(0);

String gradeMsg = switch(gradeLetter) {
    case 'A', 'B' -> "Great job";
    case 'C' -> "Good job";
    case 'D', 'F' -> "Bad job";
    default -> "Not a grade";
};
System.out.println(gradeMsg);
```

Being able to assign a value directly to a variable improves code conciseness and readability. In the old days of Java, here's how that code would have been written:

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter your grade: ");
String gradeString = keyboard.nextLine();
char gradeLetter = gradeString.toUpperCase().charAt(0);
String gradeMsg;

switch (gradeLetter) {
    case 'A':
    case 'B':
        gradeMsg = "Great job";
        break;
    case 'C':
        gradeMsg = "Good job";
        break;
    case 'D':
    case 'F':
        gradeMsg = "Bad job";
        break;
}
```

```
        default:
            gradeMsg = "Not a grade";
            break;
    }

    System.out.println(gradeMsg);
```

The above is still valid Java! It's just the old style.

Arrow Syntax

The arrow syntax (`->`) was introduced in Java 14 to simplify switch statements and eliminate the need for break statements, which were often a source of errors due to fall-through behavior:

```
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> System.out.println("6 letters");
    case TUESDAY -> System.out.println("7 letters");
    case THURSDAY, SATURDAY -> System.out.println("8 letters");
    case WEDNESDAY -> System.out.println("9 letters");
}
```

This syntax ensures that each case is **self-contained** and does not fall through to the next case, which improves code clarity and maintainability.

These enhancements make switch statements more powerful and easier to use, aligning with modern programming practices.

Yield Keyword

The `yield` keyword was introduced in Java 14 to be used within switch expressions. It allows a switch case to return a value, which can then be assigned to a variable.

```
String message = switch (number) {
    case 1 -> "Got a 1";
    case 2 -> "Got a 2";
    default -> "More than 2";
};
```

In the above example, the `yield` keyword is implicit because the arrow syntax (`->`) directly returns the value. If you need to perform more complex operations within a case, you can explicitly use `yield`:

```
String message = switch (number) {
    case 1 -> {
        // some complex logic
        yield "Got a 1";
    }
    case 2 -> {
        // some complex logic
        yield "Got a 2";
    }
    default -> {
        // some complex logic
        yield "More than 2";
    }
};
```

Printf

You can use the `System.out.printf` method to perform formatted console output.

```
System.out.printf(FormatString, ArgList);
```

format string is a string that contains text and special formatting specifiers. Arg list is optional. It is a list of additional arguments that will be formatted according to the format specifiers listed in the format string.

```
public class Main {
    public static void main(String[] args) {
        int number = 42;
        String text = "Hello, World!";
        double decimal = 3.14159;
        double grossPay = 5874.128

        System.out.printf("Integer: %d\n", number);
        System.out.printf("String: %s\n", text);
        System.out.printf("Decimal: %.2f\n", decimal);
        System.out.printf("Decimal with comma: %, .2f\n", grosspay);
    }
}
```

In the above example:

- `%d` is a placeholder for an integer.

- `%s` is a placeholder for a string.
- `%.2f` is a placeholder for a floating-point number, formatted to two decimal places.

You can have multiple FormatStrings within print F

```
System.out.printf("Hello %s! One kilobyte is %,d bytes.", "World", 1024);
```

Source and more examples here: [Java Output printf\(\) Method](#)

You can specify field width by including a number between the % and the formatter (d):

```
number = 9;
System.out.printf("The value is %6d\n", number);
the value is      9

number = (double) 9.76891
System.out.printf("The value is %6.2f\n", number);
the value is      9.77
```

In the first example, the field is 6 spaces wide. The second is as well, but four spaces are used by the output.

Aligning text using spacing description between percentage and data type signifier.

Similar to adding commas and decimals (`%,.2f`) to indicate the need for commas and the number of decimal places, you can give a number to specify the width of the string (`%10s` will give you a formatted string of length 10).

By default, the string is right-aligned, but you can make it left-aligned by writing a MINUS sign before the width (example: `%-10s`). See this example:

```
// LEFT ALIGNED
System.out.printf("%-15s: Peter %n", "name");
System.out.printf("%-15s: Bentley %n", "school");
System.out.printf("%-15s: CIS %n", "department");

name           : Peter
school         : Bentley
department     : CIS

// RIGHT ALIGNED
System.out.printf("%15s: Peter %n", "name");
System.out.printf("%15s: Bentley %n", "school");
```

```

System.out.printf("%15s: CIS %n", "department");

        name: Peter
       school: Bentley
    department: CIS

// ALIGNING MULTIPLE VALUES
System.out.printf("%-20s %,14.2f \n", "Yearly pay:", 9999999.99);
System.out.printf("%-20s %,14.2f \n", "Lifetime pay:", 9999999.99);

Yearly pay:                9,999,999.99
Lifetime pay:              99,999.99

```

See `Columns.java` and `CurrencyFormat.java`.

String format()

`String.format` in Java is used to create formatted strings without printing them directly to the console.

The `String.format` method works exactly like the `System.out.printf` method, except that it does not display the formatted string on the screen. Instead, it returns a reference to the formatted string.

You can assign the reference to a variable, and then use it later.

```
String.format(FormatString,ArgumentList);
```

The difference is that `printf` just prints the formatted string but doesn't create a new variable.

```

public class Main {
    public static void main(String[] args) {
        int number = 42;
        String text = "Hello, World!";
        double decimal = 3.14159;

        String formattedString = String.format("Integer: %d, String: %s,
Decimal: %.2f", number, text, decimal);
        System.out.println(formattedString);
    }
}

```

See examples:

- `CurrencyFormat2.java`
- `CurrencyFormat3.java`