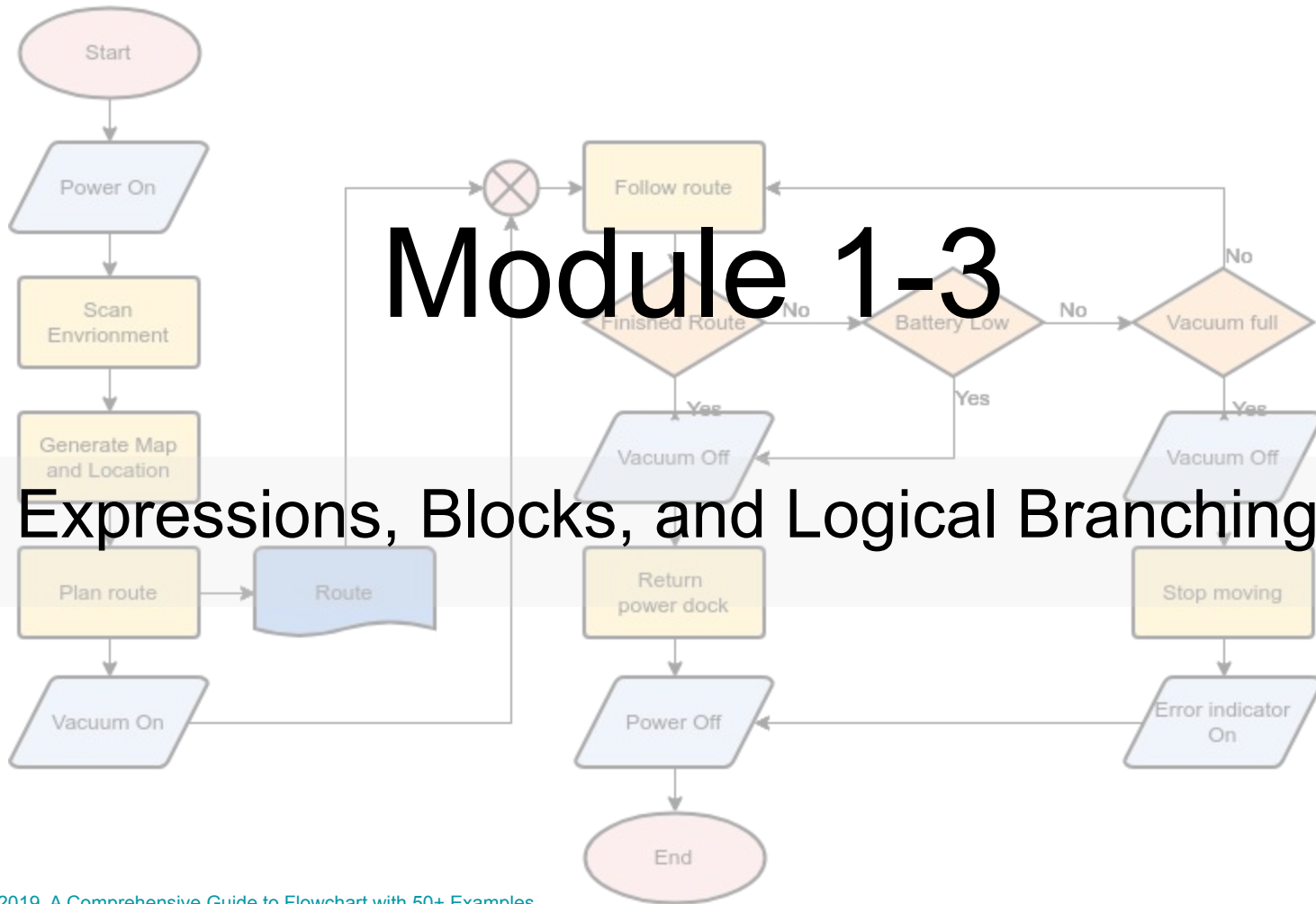


Module 1-3



Module 1 Day 3

Can/Do you?

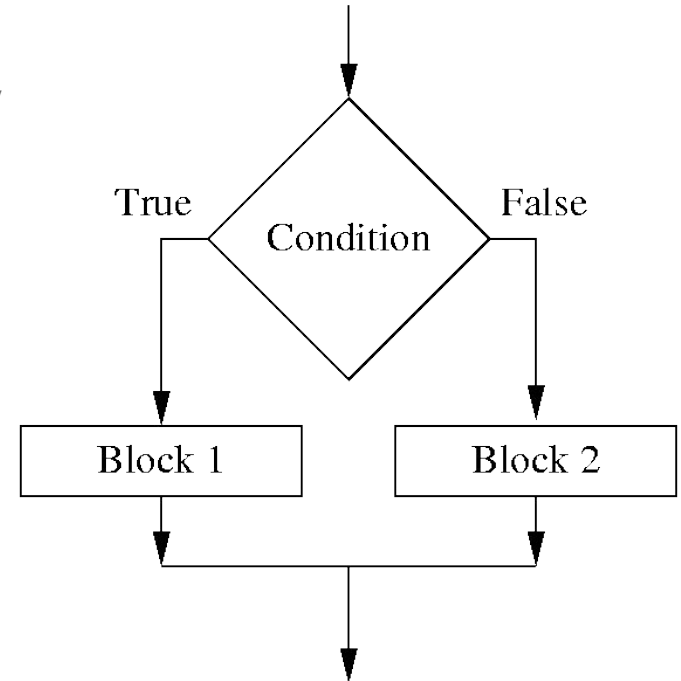
1. ... explain the things that can make up an Expression?
2. ... define what is meant by a *Statement* in a programming language?
3. ... describe the purpose and use of a *Block* in a programming language?
4. ... know what a *Boolean Expression* is and how it is used?
5. ... understand the *Comparison Operators* and how to use them?
6. ... understand the *Logical Operators* and how to use them?
7. ... explain casting/data conversion, when it occurs, and why it's used?
8. ... understand how `()` work with boolean expressions and why using them makes code more clear?
9. ... understand the Truth Table and how to use it to figure out AND and OR interactions?

Java: Expressions

- Code is made up of expressions and statements.
- Expressions evaluate to a single value.
- Computers evaluate each expression separately.
- We use balanced parentheses to control the order of evaluation or to make the order unambiguous.
 - `x + y / 100` // ambiguous
 - `(x + y) / 100` // unambiguous, recommended

Java: Expressions - Booleans

- A **Boolean Expression** is a Logical Expression that evaluates to a literal true or false.
- Boolean Expressions are used to conditionally execute blocks of code.



Java: Statements

- Statements are roughly equivalent to sentences in natural languages.
- A statement forms complete unit of execution.
- Some expressions can be made into a statement by terminating the expression with a semicolon (;).

Assignment expressions

`aValue = 8933.234;`

Any use of ++ or --

`aValue++;`

Method invocations

`System.out.println("Hello World!");`

Declaration Statements

`double currentTemp;`

Control flow statements

`for, while, do-while` (... more on these later this week)

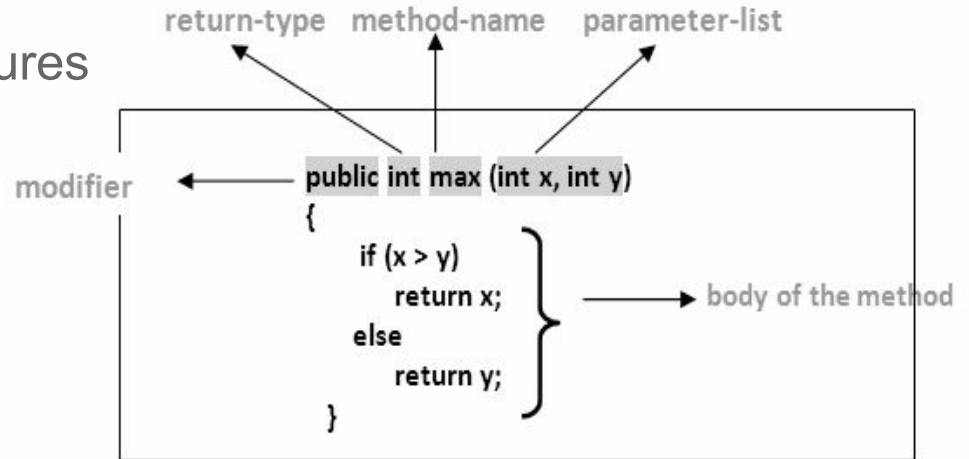
Blocks

- Code that is related (either to conform to the Java language standard or by choice) is enclosed in a set of curly braces ({ ... }). The contents inside the curly braces is known as a “block of code”, “code block” or, simply, a “block.”
- A **block** is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.
- Blocks are used in:
 - Conditional Statements (we will talk about this today)
 - Methods (ditto)
 - Loops

Additional Reading on Blocks (and Scope!) https://www.w3schools.com/java/java_scope.asp

Methods

- A method is a named block of code. It can accept multiple values and return a single value**.
 - Not required to accept values, but it can.
 - Not required to return a value, but it can.
- Methods have Method Signatures
 - Descriptive Names
 - Input Parameters



Conditional Statements

- A conditional statement allows for the execution of code only if a certain condition is met. The condition **must be, or must evaluate to a boolean value (true or false).**
- The if statement follows this pattern:

```
if (condition) {  
    // do something if condition is true.  
}  
else {  
    // do something if condition is false.  
}
```

- The else is optional... but you cannot have an else by itself without an if.
- The parenthesis around the condition if also required.

Conditional Statements

Here is an example:

```
public class Bear {  
  
    public static void main(String[] args) {  
  
        boolean isItFall = true;  
  
        if (isItFall == true) {  
            System.out.println("ok Hibernation time zzzz.");  
        }  
        else {  
            System.out.println("let's see what the humans are up to!");  
        }  
    }  
}
```

The == symbol checks for equivalence. It is not the same as =, which is for assignment.

The output of this code is “ok Hibernation time zzzz. Changing isItFall to false would cause the output to be “let’s see what the humans are up to!”

Conditional Statements

Here is an example:

```
public class Bear {  
  
    public static void main(String[] args) {  
  
        boolean isItFall = true;  
  
        if (isItFall) {  
            System.out.println("ok Hibernation time zzzz.");  
        }  
        else {  
            System.out.println("let's see what the humans are up to!");  
        }  
    }  
}
```

Since isItFall is a boolean already, using the expression ***isItFall == true*** is redundant, the preferred style is shown here.

Likewise, to negate the boolean isItFall, the preferred style is to write !isItFall as opposed to isItFall == false.

Conditional Statements

Here is another example:

```
public class Bear {  
  
    public static void main(String[] args) {  
  
        String season = "Winter";  
  
        if (season == "Winter") {  
            System.out.println("ok Hibernation time zzzz.");  
        }  
    }  
}
```

season is not a boolean, but it is used as part of a boolean expression:
Is the season equivalent to the literal string "Winter?"

The output of this code is "ok Hibernation time zzzz."

Conditional Statements

Here is a tricky example. What do you think the output is? Will it compile and run?

```
public class Bear {  
    public static void main(String[] args) {  
        boolean isWinter = false;  
  
        if (isWinter = true) {  
            System.out.println("ok Hibernation time zzzz.");  
        }  
        else {  
            System.out.println("I'm starving! Time for breakfast.");  
        }  
    }  
}
```

Conditional Statements: Comparison Operators

The following operators allow you to compare *numbers*:

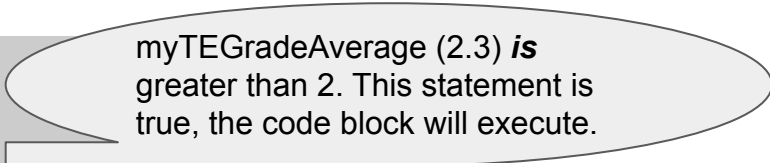
- **==** : Are 2 values equal to each other
- **!=** : Are 2 values NOT equal to each other
- **>** : Is a value greater than another value
- **<** : Is a value less than another value
- **>=** : Is a value greater or equal to another value
- **<=** : Is a value less than or equal to another value

Conditional Statements: Numerical Comparisons

Here is another example:

```
double myTEGradeAverage = 2.3;

if(myTEGradeAverage >= 2) {
    System.out.println("I am in good standing!");
}
else {
    System.out.println("I must work harder!");
}
```



myTEGradeAverage (2.3) *is* greater than 2. This statement is true, the code block will execute.

Conditional Statements : Ternary Operator

The ternary operator can be used to simplify conditional statements to assign values.

(condition to evaluate) ? //assignment if condition is true : //assignment if condition is false;

- You can assign the result of a ternary to a variable if needed. The data type of this variable would be the same as whatever the expressions on both sides of the colon (the **true case** and **false case**) resolve to.
- Conditional assignment of values to variables is the most common use of Ternary Expressions:

```
int latePenalty = ( submitDateTime > dueDate) ? 1 : 0 ;
```

Conditional Statements : Ternary Operator Example

These 2 blocks of code accomplish the same thing.

Ternary

```
double myNumber = 5;  
String divisibleBy2 = (myNumber%2 == 0) ? "Even" : "Odd";  
System.out.println(divisibleBy2);
```

If...Else

```
int myNumber = 5;  
String divisibleBy2 = "";  
  
if (myNumber%2 == 0 ) {  
    divisibleBy2 = "Even";  
}  
else {  
    divisibleBy2 = "False";  
}  
System.out.println(divisibleBy2);
```


AND / OR Operations: Combining Booleans

- A condition needs to be resolved into a true or false value, and we can achieve this by using the `==` operator.
- We can use the AND and OR conditional operators to create logical expressions that are only true if *multiple* conditions are true, or if *one of many* conditions are true
- The **AND** operator in Java is: **&&**
- The **OR** operator in Java is **||** (these are pipe symbols, it is typically located under the backspace and requires a shift).

AND / OR: Truth Table

We evaluate AND / OR using truth tables:

- For AND statement:
 - **True AND True** is **True**
 - **True AND False** is **False**
 - **False AND True** is **False**
 - **False AND False** is **False**
- For OR statement:
 - **True OR True** is **True**
 - **True OR False** is **True**
 - **False OR True** is **True**
 - **False OR False** is **False**

A	B	!A	A && B	A B	A ^ B
TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE

AND / OR: Exclusive OR

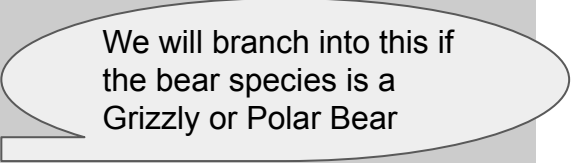
There is a third case called an “Exclusive Or” or XOR for short. The operator is the carrot symbol (\wedge).

- For XOR statements:
 - **True** XOR **True** is **False**
 - **True** XOR **False** is **True**
 - **False** XOR **True** is **True**
 - **False** XOR **False** is **False**

In most day to day programming, XOR is not used very often.

AND / OR: Examples

```
public class Bear {  
  
    public static void main(String[] args) {  
  
        String bearSpecies = "Panda";  
  
        if (bearSpecies == "Grizzly" || bearSpecies == "Polar") {  
            System.out.println("Uh Oh, I may be lunch! ");  
        }  
        else {  
            System.out.println("Whew, Panda, I'm ok.");  
        }  
    }  
}
```



We will branch into this if the bear species is a Grizzly or Polar Bear

The output of this code is “Whew, Panda, I'm ok.”

AND / OR: Examples

```
int gradePercentage = 70;  
  
if (gradePercentage >= 90) {  
    System.out.println("A");  
}  
  
if (gradePercentage >= 80 && gradePercentage < 90) {  
    System.out.println("B");  
}  
  
if (gradePercentage >= 70 && gradePercentage < 80) {  
    System.out.println("C");  
}  
  
if (gradePercentage >= 60 && gradePercentage < 70) {  
    System.out.println("D");  
}
```

70 is not greater or equal to 90.

The check is false.

Statement won't execute.

70 is not greater or equal to 80 and less than 90.

The check is false.

Statement won't execute.

70 is greater or equal to 70, and less than 80.

The check is true.

Statement will execute.

70 is not greater or equal to 60 and less than 70.

The check is false.

Statement won't execute.

AND / OR: Operations and Precedence

```
int myInteger = 2;

if(myInteger==2 && myInteger==3 || myInteger==4 || myInteger%2==0 || myInteger==6) {
    System.out.println("the combined statement is true.");
}
else {
    System.out.println("the combined statement is false.");
}
```

The output of this is “the combined statement is true.”

- We evaluate what's inside the parentheses from left to right.
- Equality operators (== and !=) take precedence over AND (&&) / OR(||).

Java's Order of Operations.... what we know now

PEMDAS (Arithmetic Rules)
Equality Operators (== and !=)
AND / OR (&&,)

Items at the top of the list take higher priority.