NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 2: 6th Feb 2024

- Expressions
- Statements
- WhiteSpaces
- Assignment Operator
- Equality Operator
- If-then Structure
- Logical AND-OR
- Ternery Operator
- Operator Precedence
- IF-Else Structure
- Method
- Method Overloading

Java’s Code Units

Writing code is similar to writing a document. It consists of special hierarchical units, which together form a whole.

These are:

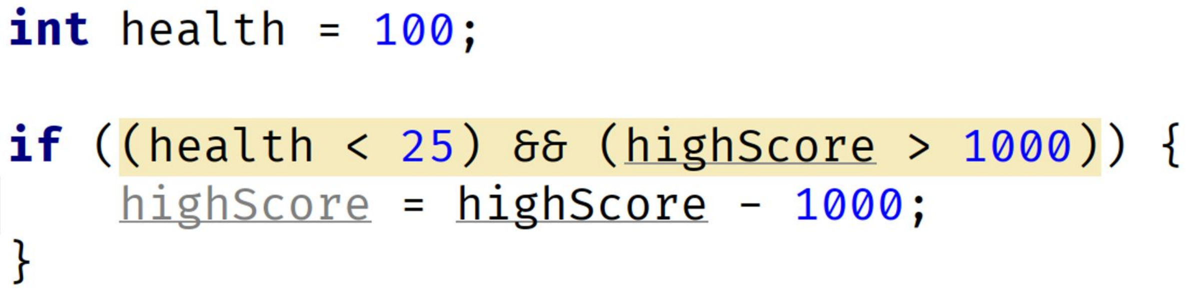
The **Expression** – An expression computes to a single value.

The **Statement** – Statements are stand alone units of work.

And **Code Blocks** – A code block is a set of zero, one, or more statements, usually grouped together in some way to achieve a single goal.

The Expression Challenge

Looking at the code below, what parts are expressions?



```
int health = 100;

if ((health < 25) && (highScore > 1000)) {
    highScore = highScore - 1000;
}
```

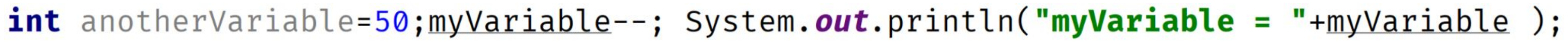
Whitespace

What is whitespace?

- Whitespace is any extra spacing, horizontally or vertically, placed around Java source code.
- It's usually added for human readability purposes.
- In Java, all these extra spaces are ignored.

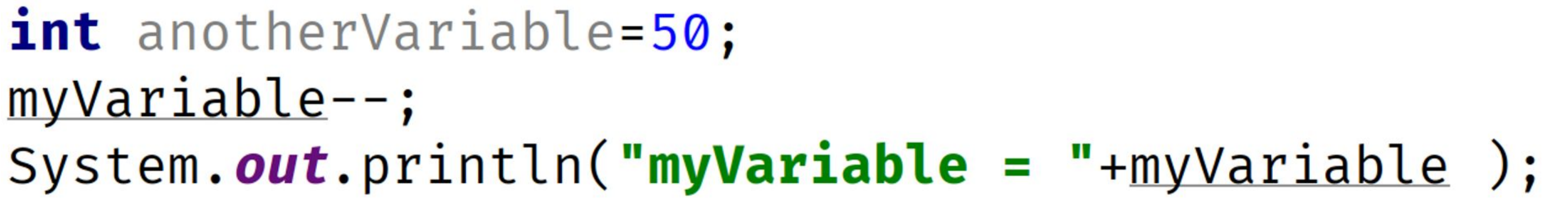
Whitespace

So Java treats code like this:



```java
int anotherVariable=50;myVariable--; System.out.println("myVariable = "+myVariable );
```

The same as code like this:



```java
int anotherVariable=50;
myVariable--;
System.out.println("myVariable = "+myVariable );
```

Whitespace Coding Conventions

Code conventions for whitespace do exist, which you can refer to for more detail.

The Google Java Style Guide, has a section on whitespace, so refer to that for more information.

if-then statements in Java

The **if-then** statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code, only if a particular tests evaluates to true.

This is known as **conditional logic**.

Conditional Logic

Conditional logic uses specific statements in Java to allow us to check a condition, and execute certain code based on whether that condition (the expression) is true or false.

Assignment Operator (=)

The assignment operator assigns the value of an expression, to the variable to the left of the operator.

```
boolean isAlien = false;
```

So, isAlien is the variable in this case, and it's been set to false, which is the value of our expression.

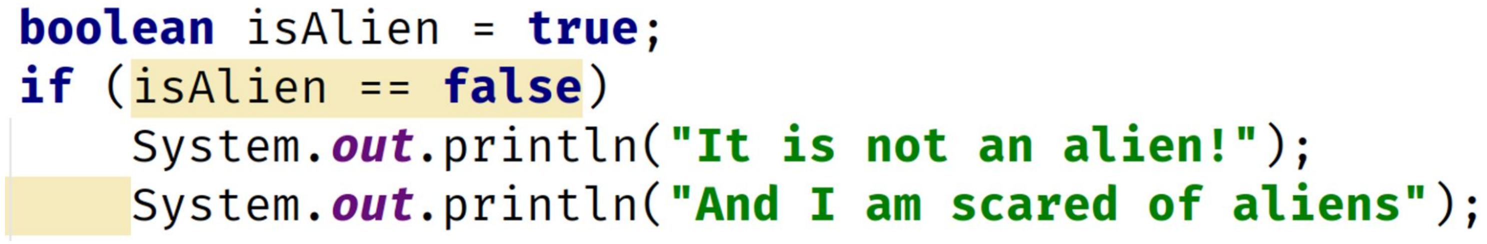
Equality Operator (==)

The equality operator tests to see if two operands are considered equal, and returns a boolean value.

```
if (isAlien == false) {
```

So, here **isAlien** is being tested against the value false.

Best Practice Rule - Always use a Code Block for If-Then statements



```
boolean isAlien = true;
if (isAlien == false)
    System.out.println("It is not an alien!");
    System.out.println("And I am scared of aliens");
```

Instead of using the if statement as we can see here, we should instead use a code block.

The Code Block

A code block allows more than one statement to be executed, in other words, a block of code.

The format is:

```
if (expression) {
    // put one or more statements here
}
```

The Logical AND operator and the Logical OR operator

The **and** operator comes in two flavours in Java, as does the **or** operator.

&& is the Logical **and** which operates on **boolean** operands – Checking if a given condition is **true** or **false**.

The **&** is a bitwise operator working at the bit level. This is an advanced concept that we won't get into here.

The Logical AND operator and the Logical OR operator

Likewise **||** is the Logical **or**, and again it operates on **boolean** operands – Checking if a given condition is **true** or **false**.

The | is a bitwise operator, which is also working at the bit level.

And just like the bitwise **and** operator, we won't be using it as much as their logical counterparts.

We'll almost always be using the logical operators.

The Ternary Operator (Condition ?: Operator)

The ternary operator – Java officially calls it the Conditional Operator – has three operands, the only operator currently in Java that does have three.

The structure of this operator is:

operand1 ? operand2 : operand3

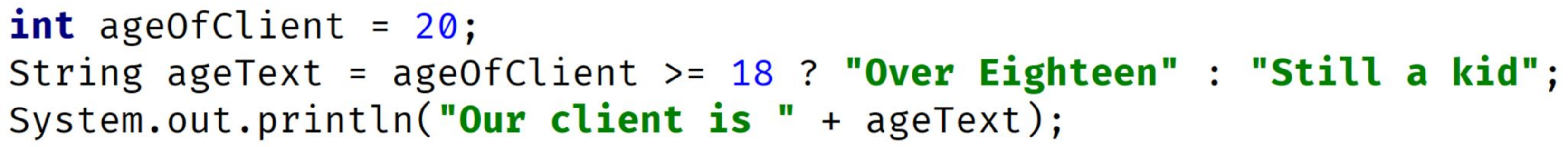
Ternary Operator ? :

The **ternary** operator is a shortcut to assigning one of two values to a variable, depending on a given condition.

So think of it as a shortcut of the **if-then-else** statement.

Ternary Operator ? :

Consider this example:



```java
int ageOfClient = 20;
String ageText = ageOfClient >= 18 ? "Over Eighteen" : "Still a kid";
System.out.println("Our client is " + ageText);
```

Operand one – **ageOfClient >= 18** in this case is the condition we're checking. It needs to return true, or false.

Operand two – "**Over Eighteen**" here is the value to assign to the variable **ageText** , if the condition above is true.

Operand three – "**Still a kid**" here is the value to assign to the variable **ageText** , if the condition above is false.

Challenge

Step 1: create a double variable with a value of 20.00.

Step 2: create a second variable of type double with a value 80.00.

Step 3: add both numbers together, then multiply by 100.00.

Step 4: use the remainder operator, to figure out what the remainder from the result of the operation in step three, and 40.00, will be.

Step 5: create a boolean variable that assigns the value true, if the remainder in step four is 0.00, or false if it's not zero.

Step 6: output the boolean variable just to see what the result is.

Step 7: write an if-then statement that displays a message, 'got some remainder', if the boolean in step five is not true.

If Then Structure

```
if (condition) {
    // Code in block will execute only if
    // condition is true

    // block can contain 1 or many statements

}
```

If with an Else block

```
if (condition) {
    // Code in block will execute only if
    // condition is true
} else {

    // Code in block will execute only if
    // condition is false

}
```

If with an Else if and Else block

```java
if (firstCondition) {
    // Code in block will execute only if
    // firstCondition is true

} else if (secondCondition) {

    // Code in block will execute if firstCondition is false
    // and secondCondition is true
```

THERE IS NO LIMIT TO THE NUMBER OF CONDITIONS THAT CAN BE TESTED

```java
} else {

    // Code in block will execute if
    // all conditions above are false
```

THE ELSE BLOCK MUST BE LAST BUT IS OPTIONAL

```java
}
```

if then else Challenge

Insert a code segment after the code we've just reviewed:

- Set the existing score variable to 10,000.
- Set the existing levelCompleted variable to 8.
- Set the existing bonus variable to 200.
- Use the same if condition (meaning if gameOver is true) you want to perform the same calculation, and print out the value of the finalScore variable.

The Method

Java's description of the method is:

A method declares executable code that can be invoked, passing a fixed number of values as arguments.

The Benefits of the Method

A method is a way of reducing code duplication.

A method can be executed many times with potentially different results, by passing data to the method in the form of arguments.

Structure of the Method

One of the simplest ways to declare a method is shown on this slide.

This method has a name, but takes no data in, and returns no data from the method (which is what the special word void means in this declaration).

```
public static void methodName() {

    // Method statements form the method body

}
```

Executing a Method as a Statement

To execute a method, we can write a statement in code, which we say is calling, or invoking, the method.

For a simple method like calculateScore, we just use the name of the method, where we want it to be executed, followed by parentheses, and a semi-colon to complete the statement.

So for this example, the calling statement would look like the code shown here:

```
calculateScore();
```

Structure of the Method

Where we previously had empty parentheses after the method name, we now have method parameters in the declaration.

```
public static void methodName(p1type p1, p2type p2, {more}) {

    // Method statements form the method body

}
```

Parameters or Arguments?

Parameters and arguments are terms that are often used interchangeably by developers.

But technically, a parameter is the definition as shown in the method declaration, and the argument will be the value that's passed to the method when we call it.

Executing a Method with parameters

To execute a method that's defined with parameters, you have to pass variables, values, or expressions that match the type, order and number of the parameters declared.

In the calculateScore example, we declared the method with four parameters, the first a boolean, and the other three of int data types.

So we have to pass first a boolean, and then 3 int values as shown in this statement:

```
calculateScore(true, 800, 5, 100);
```

We can't pass the boolean type in any place, other than as the first argument, without an error.

Executing a Method with parameters

The statement below would cause an error.

```
calculateScore(800, 5, 100, true);
```

And you can't pass only a partial set of parameters as shown here.

This statement, too, would cause an error.

```
calculateScore(true, 800);
```

Method structure with parameters and return type

So, similar to declaring a variable with a type, we can declare a method to have a type.

This declared type is placed just before the method name.

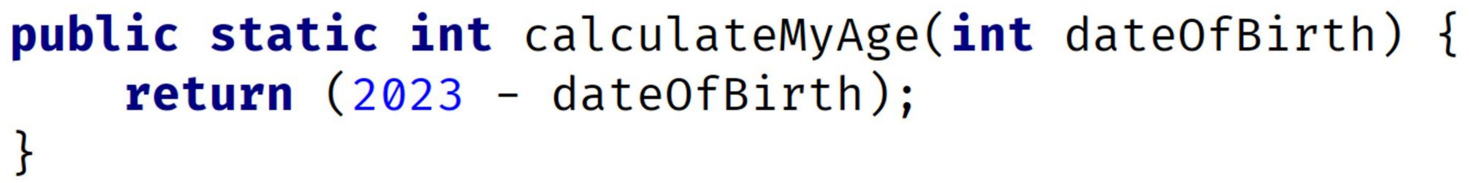
In addition, a return statement is required in the code block, as shown on the slide, which returns the result from the method.

```
// Method return type is a declared data type for the data that
// will be returned from the method
public static dataType methodName(p1type p1, p2type p2, {more}) {

    // Method statements
    return value;

}
```

Method structure with parameters and return type

An example of a method declaration with a return type is shown here.

In this case, the return type is an int.



```java
public static int calculateMyAge(int dateOfBirth) {
    return (2023 - dateOfBirth);
}
```

This method will return an integer when it finishes executing successfully.

The return statement

So, what's a return statement?

Java states that a return statement returns control to the invoker of a method.

The most common usage of the return statement, is to return a value back from a method.

In a method that doesn't return anything, in other words a method declared with void as the return type, a return statement is not required.

But in methods that do return data, a return statement with a value is required.

Is the method a statement or an expression?

A method can be a statement or an expression in some instances.

Any method can be executed as a statement.

A method that returns a value can be used as an expression, or as part of any expression.

What are functions and procedures?

Some programming languages will call a method that returns a value, a function, and a method that doesn't return a value, a procedure.

You'll often hear function and method used interchangeably in Java.

The term procedure is somewhat less common, when applied to Java methods, but you may still hear a method with a void return type, called procedure.

Declaring the Method

So there are quite a few declarations that need to occur as we create a method.

This consists of:

- Declaring Modifiers. These are keywords in Java with special meanings, we've seen **public** and **static** as examples, but there are others.
- Declaring the return type.
  - **void** is a Java keyword meaning no data is returned from a method.
  - Alternatively, the return type can be any primitive data type or class.
  - If a return type is defined, the code block must use at least one return statement, returning a value, of the declared type or comparable type.

Declaring the Method

- Declaring the method name. Lower camel case is recommended for method names.
- Declaring the method parameters in parentheses. A method is not required to have parameters, so a set of empty parentheses would be declared in that case.
- Declaring the method block with opening and closing curly braces. This is also called the method body.

Declaring the Parameters

- Parameters are declared as a list of comma-separated specifiers, each of which has a parameter type and a parameter name (or identifier).
- Parameter order is important when calling the method.
- The calling code must pass arguments to the method, with the same or comparable type, and in the same order, as the declaration.
- The calling code must pass the same number of arguments, as the number of parameters declared.

Declaring the Return Type

When declaring a return type:

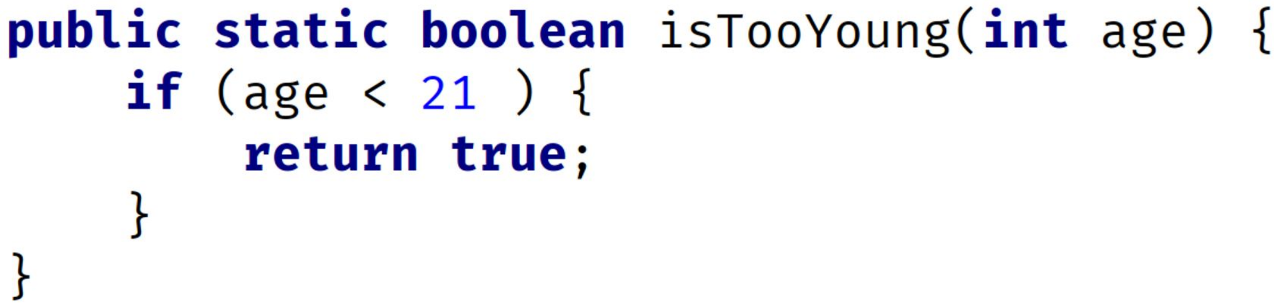
void is a valid return type, and means no data is returned.

Any other return type requires a return statement, in the method code block.

The Return Statement for methods that have a return type

If a method declares a return type, meaning it's not void, then a return type is required at any exit point from the method block.

Consider the method block shown here:

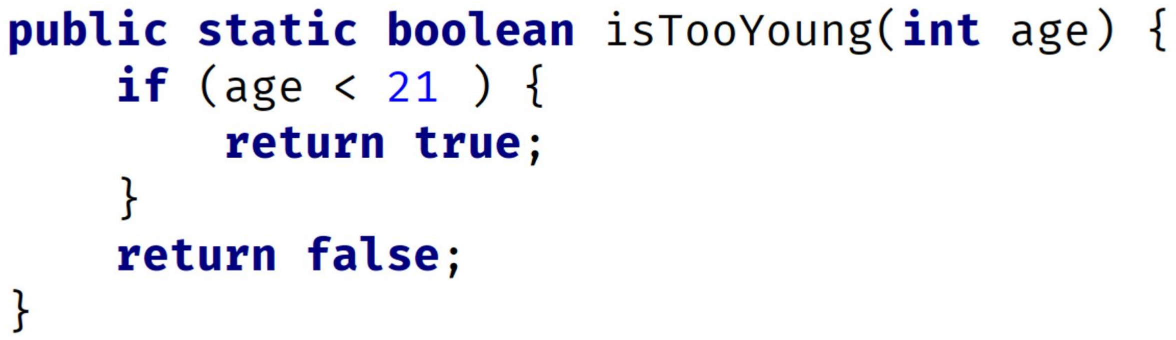


```
public static boolean isTooYoung(int age) {
    if (age < 21 ) {
        return true;
    }
}
```

The Return Statement for methods that have a return type

So in the case of using a return statement in nested code blocks in a method, all possible code segments must result in a value being returned.

The following code demonstrates one way to do this:



```java
public static boolean isTooYoung(int age) {
    if (age < 21 ) {
        return true;
    }
    return false;
}
```

The Return Statement for methods that have a return type

One common practice is to declare a default return value at the start of a method, and only have a single return statement from a method, returning that variable, as shown in this example method:

```
public static boolean isTooYoung(int age) {
    boolean result = false;
    if (age < 21 ) {
        result = true;
    }
    return result;
}
```

The Return Statement for methods that have void as the return type

The return statement can return with no value from a method, which is declared with a void return type.

In this case, the return statement is optional, but it may be used to terminate execution of the method, at some earlier point than the end of the method block, as we show here:

```java
public static void methodDoesSomething(int age) {
    if (age > 21) {
        return;
    }
    // Do more stuff here

}
```

The Method Signature

A method is uniquely defined in a class by its name, and the number and type of parameters that are declared for it.

This is called the method signature.

You can have multiple methods with the same method name, as long as the method signature (meaning the parameters declared) are different.

Default values for parameters

In many languages, methods can be defined with default values, and you can omit passing values for these when calling the method.

But Java doesn't support default values for parameters.

There are work-arounds for this limitation, and we'll be reviewing those at a later date.

But it's important to state again, in Java, the number of arguments you pass, and their type, must match the parameters in the method declaration exactly.

Revisiting the main method

Now, that we're armed with knowledge about methods, we can revisit the main method, and examine it again.

The main method is special in Java, because Java's virtual machine (JVM) looks for the method, with this particular signature, and uses it as the entry point for execution of code.

```java
public static void main(String[] args) {
    // code in here
}
```

Method Challenge

In this challenge we're going to create two methods:

The first method should be named displayHighScorePosition.

This method should have two parameters, one for a player's name, and one for a player's position in a high score list.

This method should print a message like "Tim managed to get into position 2 on the high score list".

Method Challenge

The second method should be named calculateHighScorePosition.

This method should have only one parameter, the player's score.

This method should return a number between 1 and 4, based on the score values shown in this table.

Score	**Result**
Score greater than or equal to 1000	1
Score greater than or equal to 500 but less than 1000	2
Score greater than or equal to 100 but less than 500	3
All other scores	4

Finally, we'll call both methods and display the results for the following scores: 1500, 1000, 500, 100, and 25.

Method Overloading

Method overloading occurs when a class has multiple methods, with the same name, but the methods are declared with different parameters.

So you can execute a method with one name, but call it with different arguments.

Java can resolve which method it needs to execute, based on the arguments being passed, when the method is invoked.

More on Method Signatures

A method signature consists of the name of the method, and the uniqueness of the declaration of its parameters.

In other words, a signature is unique, not just by the method name, but in combination with the number of parameters, their types, and the order in which they are declared.

A method's return type is not part of the signature.

A parameter name is also not part of the signature.

Valid Overloaded Methods

The type, order, and number of parameters, in conjunction with the name, make a method signature unique.

A unique method signature is the key for the Java compiler, to determine if a method is overloaded correctly.

The name of the parameter is not part of the signature, and therefore it doesn't matter, from Java's point of view, what we call our parameters.

Valid Overloaded Methods

```java
public static void doSomething(int parameterA) {
    // method body
}

public static void doSomething(float parameterA) {
    // method body
}

public static void doSomething(int parameterA, float parameterB) {
    // method body
}

public static void doSomething(float parameterA, int parameterB) {
    // method body
}

public static void doSomething(int parameterA, int parameterB, float parameterC) {
    // method body
}
```

Invalid Overloaded Methods

Parameter names are not important when determining if a method is overloaded.

Nor are return types used when determining if a method is unique.

```java
public static void doSomething(int parameterA) {
    // method body
}

public static void doSomething(int parameterB) {
    // method body
}

public static int doSomething(int parameterA) {
    return 0;
}
```

Overloaded Method Challenge Instructions

Create two methods with the same name: convertToCentimeters

The first method has one parameter of type int, which represents the entire height in inches. You'll convert inches to centimeters, in this method, and pass back the number of centimeters, as a double.

The second method has two parameters of type int, one to represent height in feet, and one to represent the remaining height in inches. So if a person is 5 foot, 8 inches, the values 5 for feet and 8 for inches would be passed to this method. This method will convert feet and inches to just inches, then call the first method, to get the number of centimeters, also returning the value as a double.

Both methods should return a real number or decimal value for total height in centimeters.

Call both methods, and print out the results.

The conversion formula from inches to centimeters is 1 inch = 2.54 cm.

Also, remember one foot = 12 inches.