

Introduction

Welcome to Section 5, and congratulations on making it so far.

Tim Buchalka here again. This next section is dealing with methods. They are a way to group code together, similar to code blocks, but with a lot more flexibility and power.

We will take the knowledge you have learned about expressions, statements and code blocks to the next level, combining them with the power of methods.

Let's make a start.

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:

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

The same as code like this:

```
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 which was seen previously in this course, has a section on whitespace, so refer to that for more information, and the link to that is again in the resources section of this video.

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

```
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

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

THE ELSE BLOCK MUST BE LAST BUT IS OPTIONAL

```
}
```

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, then 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, I 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);
```

I 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

```
// 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;  
  
}
```

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 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.

```
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. It is assumed and execution is returned after the last line of code in the method is executed.

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

The Method

Java's documentation states that:

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

Is the method a statement or an expression?

Like some of the abbreviated operators we learned about, 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

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

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:

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

The Return Statement for methods that have a return

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

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 shown here:

```
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.

This will become important later in this section, when we cover overloaded methods.

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.

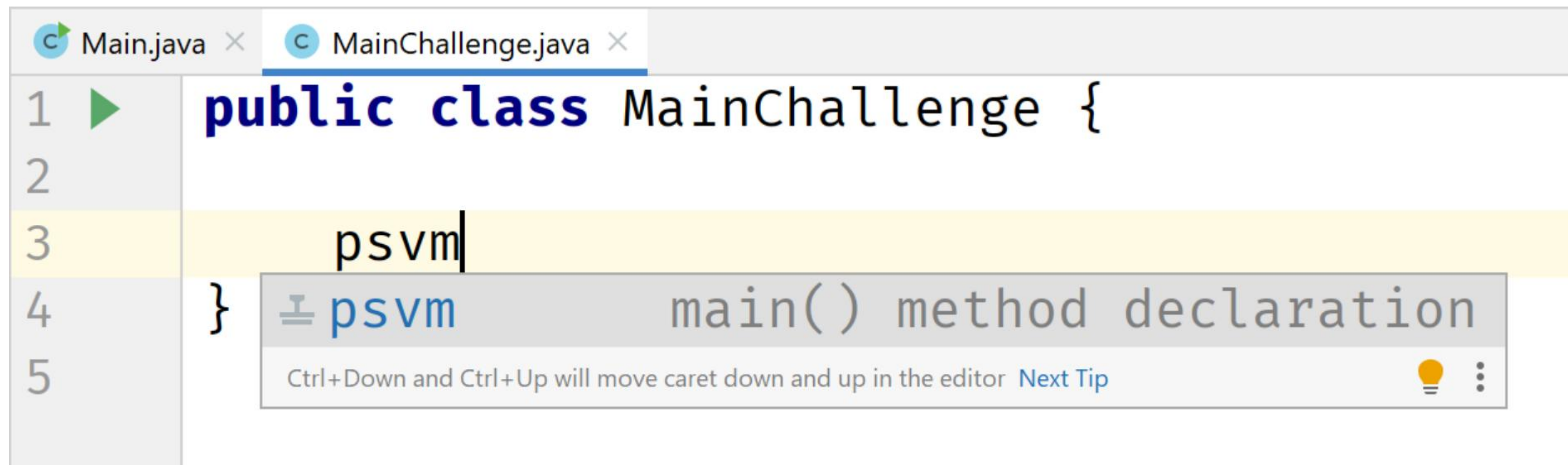
The main method is special in Java, because Java's virtual machine (JVM) looks for the method, with this particular signature, and uses it as the entry point for execution of code.

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


IntelliJ hint

Finally, in IntelliJ, if you type `psvm` and hit enter, IntelliJ will insert the main method signature as we show here.

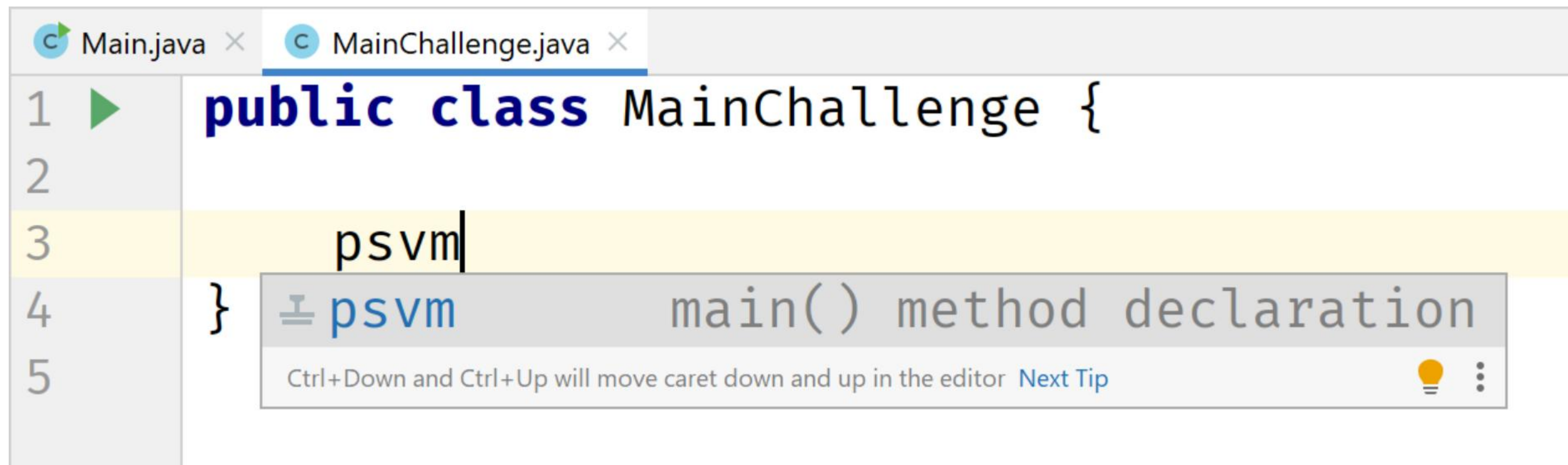
The only reason to memorize this signature, would be if you were taking a certification exam.



IntelliJ hint

Finally, in IntelliJ, if you type `psvm` and hit enter, IntelliJ will insert the main method signature as we show here.

The only reason to memorize this signature, would be if you were taking a certification exam.



The screenshot shows the IntelliJ IDEA IDE with two tabs: 'Main.java' and 'MainChallenge.java'. The 'MainChallenge.java' tab is active. The code editor shows the following code:

```
1 public class MainChallenge {  
2  
3     psvm  
4 }  
5
```

A tooltip is displayed over the 'psvm' text on line 3. The tooltip contains the text 'psvm' followed by 'main() method declaration'. Below this, a hint message reads: 'Ctrl+Down and Ctrl+Up will move caret down and up in the editor Next Tip'. The hint message is accompanied by a lightbulb icon and a vertical ellipsis icon.

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 `calculateHighScorePosition` with the following scores:
1500, 1000, 500, 100, and 25.

Method Challenge

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

So our ranges are shown here, so let's write some code that does this.

Code Comparison in IntelliJ

First, remember that my solution will be in a zip file in the resources section of each lecture.

This zip file is the entire IntelliJ project zipped up.

Second, I'll talk about an IntelliJ feature that will help you compare your code to mine.

Finally, there is another feature that lets you look at the history of your own versions in IntelliJ, in case you had something working, broke it, and want to get back that old version.

Coding Exercises

Let's discuss a new feature you will see in the course moving forward. The **coding exercise**.

Coding exercises are a feature added by Udemy, to allow instructors to add exercises that students can complete on their own.

The goal of these exercises is to help reinforce concepts you've been taught, and to encourage you to share your solutions with other students in the course, to start a dialogue about different ways to solve an exercise.

You'll need to understand the problem, the specific requirements, and then design and code your own solution.

Coding Exercises

You'll find them in the course sections going forward, including this one, and they will show in the course curriculum marked as a "Coding Exercise".

There are over fifty already in the course. And I have plans to add more when time permits. They have been in the course for years. However, recently, Udemy has updated coding exercises to fully Java 17 which, as you know, is the version of Java I currently recommend, and the version used in this course.

The good news is all coding exercises have been updated to this new format.

The cool thing about coding exercises is that you can literally click a button, and have your solution checked immediately.

Coding Exercises

I'll give you a coding exercise to complete, and you can type in your solution to it, interactively on the screen.

When you think you've coded it correctly, you can click a button, and see if the solution is correct.

This is different to the challenges that you've seen so far in the course, where I show you the requirements on the screen, and then ask you to pause the video to solve the solution on your own, but then I walk through the solution with you.

Coding exercises are different, in that I give you the exercise, then you do them without seeing a solution in a video, but you do have that button to click to check the answer. Plus you can review the solutions that other students have posted, and post your solution for others to see.

Coding Exercises

There's still plenty of upcoming challenges in this course where I do show solutions in a video. Think of coding exercises as another way to challenge yourself, as well as being able to put into practice what has been taught in a particular section of the course.

I think you'll find the coding exercises are a lot of fun, once you learn some tricks to doing them.

Coding exercises might be a bit hard to understand initially.

Coding Exercises

In this video, we're going to walk through a sample coding exercise together, step by step. This will be in the new coding exercise format I mentioned.

This is the only time I'll do this, and the purpose of this video is to help you be successful in doing the rest of these coding exercises on your own.

The easiest way for you to follow along would probably be to have this video playing in one browser tab and have the coding exercise pulled up side-by-side in another browser tab.

Note that I have not mentioned IntelliJ. Unlike other coding in this course, you won't need IntelliJ for coding exercises, you'll do everything in the browser, on Udemy's website.

Ok lets make a start. As you can see I have the coding exercise open.

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 multiple methods with the same 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

This slide demonstrates some valid overloaded methods, for the `doSomething` method.

```
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.

```
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`.

Overloaded Method Challenge Instructions

- 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 $1 \text{ inch} = 2.54 \text{ cm}$.

Also, remember one foot = 12 inches.

You can use the link below to test your results:

<https://www.metric-conversions.org/length/feet-to-centimeters.htm>

Seconds And Minutes Challenge

In this challenge, we're going to create a method, that takes time, represented in seconds, as the parameter.

We'll then want to transform the seconds into hours.

Next, you'll display the time in hours with the remaining minutes and seconds in a String.

We'll do this transformation in two steps, which allows us to use overloaded methods.

Seconds And Minutes Challenge

We want to create two methods with the same name: `getDurationString`

- The first method has one parameter of type `int`, named `seconds`.
- The second method has two parameters, named `minutes` and `seconds`, both `ints`.
- Both methods return a `String` in the format shown:

'XXh YYm ZZs'

where `XX` represents the number of hours, `YY` the number of minutes, and `ZZ` the number of seconds.

- The first method should in turn call the second method to return its results.

Seconds And Minutes Challenge Tips

- Make both methods public and static as we've been doing so far in this course.
- Remember that one minute is 60 seconds, and one hour equals 60 minutes, or 3600 seconds.
- Start by creating a new project, and call it SecondsAndMinutesChallenge.

Seconds And Minutes Challenge Bonus

Add validation to the methods as a bonus:

- For the first method, the seconds parameter should be ≥ 0 .
- For the second method, the minutes parameter should be ≥ 0 , and the seconds parameter should be ≥ 0 , and ≤ 59 .
- If either method is passed an invalid value, print out some type of meaningful message to the user.

Seconds And Minutes Challenge Bonus:

In this part of the challenge, we'll add validation to the methods as a bonus:

- For the first method,
 - the seconds parameter should be ≥ 0 .

For the second method,

- the minutes parameter should be ≥ 0 .
 - and the seconds parameter should be ≥ 0 and ≤ 59 .
- If either method is passed an invalid value, print out some type of meaningful message to the user.