At promised, here are the slides for the Java Masterclass course re-master (that really is a mouthful).

**December 2019 update**

This is an update to the progress made - This document is now over 130 pages and growing fast. My recent 8 week illness, plus work on my new Java 11 Certification course (now live) contributed to slower than expected progress, but a good portion of work has been done. I'm taking a few weeks break and will be ready to start fresh in 2020 where I expect progress to get a lot faster. I am aiming to get the bulk of the remastering complete in the early months of 2020.

Make sure you check out the work in progress section at the end of the course where you will always find the latest remastered content, and also note that only the slides for the remastered portions of the course are in this document. Why? Because I never created slides for the older videos, they just got "baked" into the videos. So as each video is re-recorded new slides are created and added to this document.

**What is being remastered?**

Well first, some history.

I originally conceived the idea of this Java Masterclass in 2015 after students in my Android Java course kept asking me to extend the size of the Java tutorial section in that course.

Realising that Java really needed to be a complete course in it's own right, I created the Java Masterclass course and it got released to rave reviews late in 2015 and has been improved and updated ever since. Over 391,000 students have been through the course, and left 92,000+ reviews. It's grown into a huge course of over 80 hours of video training.

Like any educational material, over time bugs are found, things need updating, and combining this with what I have learnt about producing courses and using the feedback I have received from students like you lead me to making a decision in July 2019 to re-record the entire course. It's got past the point when I can just release an update video here and there.

**What is getting re-recorded?**

I am literally re-recording each and every video in the course. While the content will be substantially the same, there will be improvements. Firstly to the video recording quality itself. One of the issues students found was the font size. I'm using much larger fonts.

Secondly these slides (now part of many videos) help you understand key concepts.

Third, in the case of certain videos that did not adequately explain the concepts covered to the level I would have liked, they are being enhanced with additional clarification and explanation.

The end result will be a much improved course that is really going to help you become a Java Programmer, better than ever.

I sincerely believe this is the best content I have ever produced, and I'm looking forward to getting the content released. The fact that you are reading this is a sign that I have already released some of this new content on Udemy.

**When will the remaster be completed?**

Keep in mind that it will take me some time to get the new content completed. As explained in the early videos in the course, I'm releasing the new content one section at a time. This is the best way to reduce confusion between old and new content.

As new content is released this document will be updated. I'll be sending out a regular email to everyone who requested access to the slides.

**How do you get notifications about new content and new slides?**

I'll be sending out regular updates to the email list you used to get access to these slides. Just watch for regular emails letting you know what content is available (both new videos and updated slides).

**Are you a pirate? (no judgement).**

If you happened to download these slides and/or my course from the Internet, keep in mind that what you have is likely very out of date (both slides and videos). I won't judge your actions other than to say that I have spent hundreds of hours producing and supporting this course, and if you find the content valuable I would appreciate you buying your own copy of it.

Please consider clicking this link to get your very own copy of this course at the cheapest price I can make it available.

Unfortunately I have to do my part to make it harder for people to illegally download my content. This is my full time "job" and if you've seen a picture of me you know I like to eat. Staying on this email list is the best way to be notified about new content.

## Current version

The current version of this slides document is v1.013 created on 21st December, 2019.

## Programming Tips and Career advice

I have a Youtube channel that I post regular videos to. In the past three months there have been 100 programming tips and career advice related videos released. Having been a programmer for 35 years I know a thing or two about programming and the industry so check out my playlist of Programming Tips and Career Advice Videos.

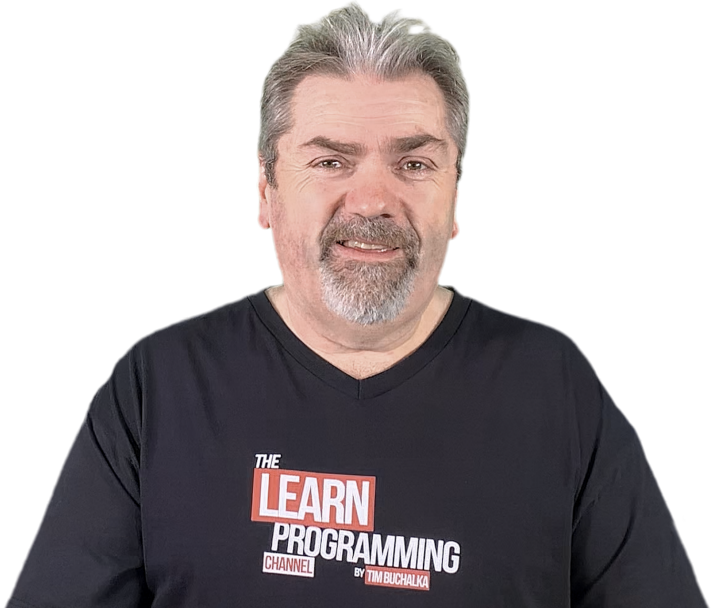
**Summary**

Well that's it for me. If you want to drop me a line you can contact me at my blog. Visit My Personal Blog here.

I hope you enjoy the course and these slides.

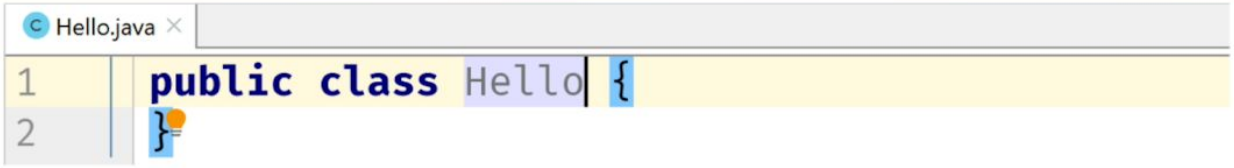
Regards

Tim Buchalka



# Main Course Slides.

# Keywords in Java



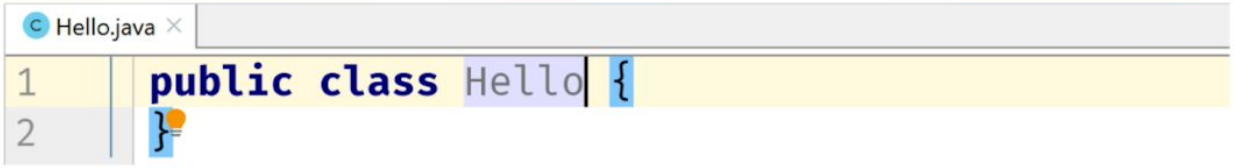

Java programs (and most other programming languages) have keywords. Each has a specific meaning and sometimes they need to be used in specific orders.

You write Java programs by following a specific set of rules, using a combination of these keywords and other things you will see which collectively form a Java program.

NOTE: keywords are case sensitive - public and Public and even PUBLIC are different things.

**public** and **class** are two java keywords - they have a specific meaning which we will find out more about moving forward.

# Access Modifiers

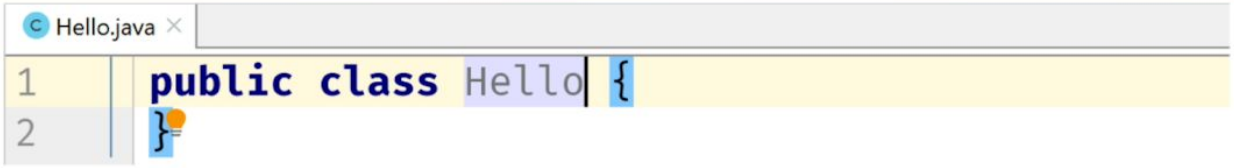



The **public** Java keyword is an access modifier. An access modifier allows us to define the scope or how other parts of your code or even some else's code can access this code. More on that later.

For now, we will use the public access modifier to give full access. We'll come back to access modifiers once we get further into the course.

Defining a class. The **class** keyword is used to define a class with the name following the keyword - Hello in this case and left and right curly braces to define the class block.

# Defining a class




To define a class requires an optional access modifier, followed by class, followed by the left and right curly braces.

The left and right curly braces are defining the class body - anything in between them is "part" of this class.  We can have data and code as you will see as we progress.

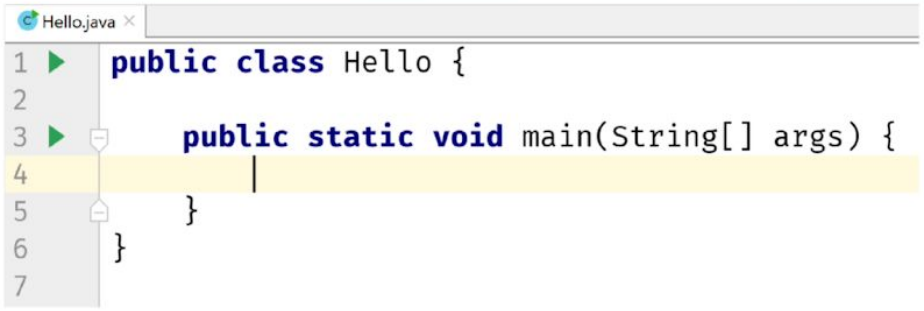
So that's our first class defined.

# Methods in Java

What is a method? It's a collection of statements (one or more) that perform an operation. We'll be using a special method called the main method that Java looks for when running a program. It's the entry point of any Java code.

You can create your own methods as you will see later, but for now, let's create this main method.

# Defining the Main Method

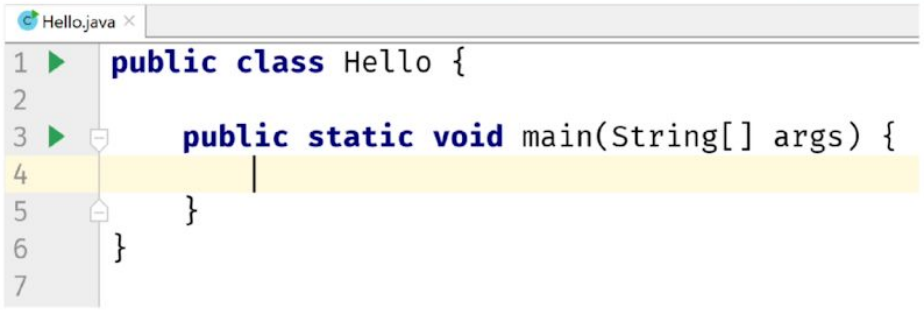



**public** is an access modifier we discussed when defining the class in the last video - same principle.

**static** is a Java keyword that needs an understanding of other concepts, for now, know that we need to have static for java to find our method to run the code we are going to add.

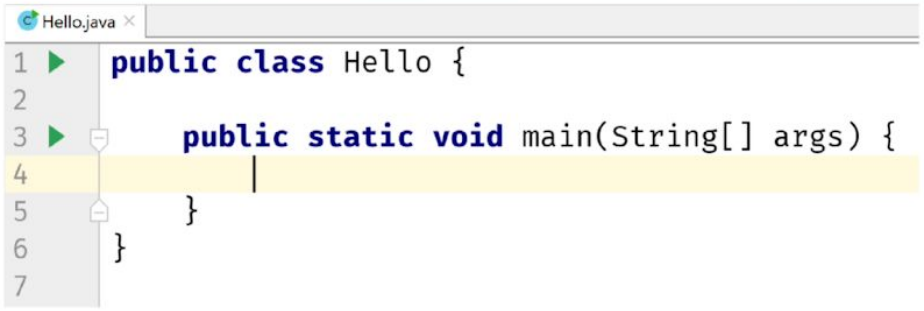
**void** is yet another java keyword used to indicate that the method will not return anything - more on that later.

# Defining the Main Method - 2




The left and right parenthesis in a method declaration are mandatory and can optionally include one or more parameters - which is a way to pass information to a method. More on that later.

# Code Blocks




**Code block** - A code block is used to define a block of code. It's mandatory to have one in a method declaration and it's here where we will be adding statements to perform certain tasks.

# Statements in Java

```
3     public static void main(String[] args) {
4         System.out.println("Hello World");
5     }
```

**Statement** - This is a complete command to be executed and can include one or more expressions (we'll see these in action later).

Here we are using the build in Java functionality to print out our message.
System.out.println is calling a Java method to print our message.

# What are Variables?

Variables are a way to store information in our computer.  Variables that we define in a program can be accessed by a name we give them, and the computer does the hard work of figuring out where they get stored in the computers random access memory (RAM).

A variable, as the name suggests can be changed, in other words, it's contents are variable.

What we have to do is tell the computer what type of information we want to store in the variable, and then give it a name.

# Data Types

There are lots of different types of data we can define for our variables.  Collectively these are known as Data types. As you may have guessed, data types are keywords in Java.

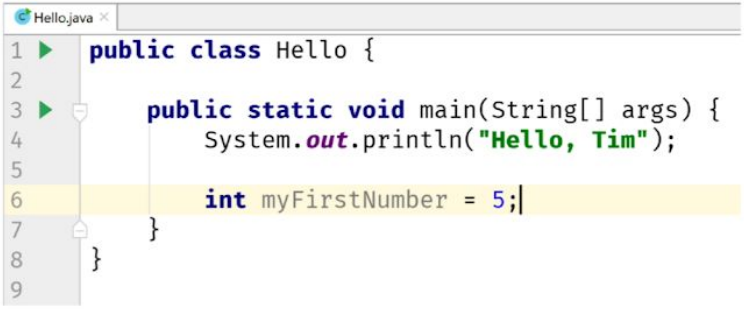
Let's start out by defining a variable of type **int** - **int** being an abbreviation for integer, a whole number (that is a number without any decimal points).

To define a variable we need to specify the data type, then give our variable a name, and optionally add an expression to initialize the variable with a value.

Let's swing back to the code to see how to define one.

# Declaration Statement

Used to define a variable by indicating the data type, and the name, and optionally to set the variable to a certain value.




Here the type (short for Data type) is an **int**, the name is **myFirstNumber** and the value we are assigning or initializing our variable with is **5**. We are declaring a variable of type **int** with the name **myFirstNumber** and assigning the value **5** to it.

# Expression

This is a construct that evaluates to a single value - we'll discuss this in much more detail in the upcoming section on expressions.

# String Literal

```
6           int myFirstNumber = 5;
7           System.out.println("myFirstNumber");
8       }
```

Any sequence of characters surrounded by double quotes is a String literal in Java. It's value cannot be changed, unlike a variable.

# Java Operators

**Java operators** or just operators perform an operation (hence the word) on a variable or value. **+**, **-** , **/** and ***** are four common ones that I feel sure you are familiar with.

There are lots more you will learn as we go through the course.

# Primitive Types

In Java primitive types are the most basic data types. The **int** is one of eight primitive types.

The eight primitive data types in Java are **boolean**, **byte**, **char**, **short**, **int**, **long**, **float** and **double**. Consider these types as the building blocks of data manipulation. Let's explore the eight primitive types in Java.

# Java Packages

A package is a way to organize your Java projects. For now, consider them as folders with **learnprogramming** in our example being a subfolder of **academy**. Companies use their domain names reversed.

So **learnprogramming.academy** becomes **academy.learnprogramming** - we will go into a lot more detail about packages later in the course.

# Wrapper classes

Java uses the concept of a Wrapper class for all eight primitive types - In the case of an **int**, we can use **Integer**, and by doing that it gives us ways to perform operations on an **int**.

```
int myMinIntValue = Integer.MIN_VALUE;
int myMaxIntValue = Integer.MAX_VALUE;
System.out.println("Integer Minimum Value = " + myMinIntValue);
System.out.println("Integer Maximum Value = " + myMaxIntValue);
```

In this case, we are using the **MIN_VALUE** and **MAX_VALUE** to get Java to tell us the minimum and maximum ranges of numbers that can be stored.

# Overflow and Underflow in Java

If you try and put a value larger than the maximum value in Java, or a value smaller than the minimum value in Java, then you will get an Overflow in the case of the maximum value or an Underflow in the case of the minimum.

The computer just skips back to the minimum number or the maximum number, which is usually not what you want.  It's an important concept to be aware of.

# Size of Primitive Types and Width

A **Byte** occupies 8 bits. A bit is not directly represented in a primitive type - We have a **boolean** which is not really the same thing that we will discuss in a future video. So a **Byte** occupies **8 bits**. We say that a **byte** has a width of **8**.

A **short** can store a large range of numbers and occupies **16 bits**, and has a width of **16**.

An **int**, has a much larger range as we know, and occupies **32 bits**, and has a width of **32**.

The point here is that each primitive type occupies a different amount of memory - we can see that an **int** needs four times the amount of space, than a **byte** does for example.

It's not particularly relevant for you to know these numbers, but it could come up as an interview question and it is useful to know that certain data types take up more space than others.

# Casting in Java

Casting means to treat or convert a number from one type to another. We put the type we want the number to be in parenthesis like this:

**(byte)** (myMinByteValue/2);

Other languages have casting, this is not just a Java thing.

# Floating Point Numbers

Unlike whole numbers, floating point numbers have fractional parts that we express with a decimal point. **3.14159** is an example.

Floating point numbers are also known as real numbers. We use a floating point number when we need more precision in calculations.

# Single and Double Precision

Precision refers to the format and amount of space occupied by the type. Single precision occupies **32 bits** (so has a width of **32**) and a Double precision occupies **64 bits** (and thus has a width of **64**).

As a result the **float** has a range from **1.4E-45** to **3.4028235E+38** and the **double** is much more precise with a range from **4.9E-324** to **1.79769313486231570E+308**.

# Floating Point Number Precision Tips

In general **float** and **decimal** are great for general floating point operations. But both are not great to use where precise calculations are required - this is due to a limitation with how floating point numbers are stored, and not a Java problem as such.

Java has a class called **BigDecimal** that overcomes this. We will talk more about that later in the course. For now just keep in the back of your mind that when precise calculations are necessary, such as when performing currency calculations, floating-point types should not be used.

But for general calculations **float** and **double** are fine. Again, we will discuss it later in the course.

# Char Data Type

A **char** occupies two bytes of memory, or **16 bit**s and thus has a width of **16**. The reason it's not just a **single byte** is that it allows you to store Unicode characters.

# Unicode

**Unicode** is an international encoding standard for use with different languages and scripts, by which each letter, digit, or symbol is assigned a unique numeric value that applies across different platforms and programs.

In the English alphabet, we have the letters A through Z. Meaning only 26 characters are needed in total to represent the entire English alphabet. But other languages need more characters, and often a lot more.

**Unicode** allows us to represent these languages and the way it works is that by using a combination of the **two bytes** that a **char** takes up in memory it can represent and one of **65535** different types of characters.

# Boolean Primitive Type

A **boolean** value allows for two choices **True** or **False**, **Yes** or **No**, **1** or **0**. In Java terms we have a **boolean** primitive type and it can be set to two values only. **true** or **false**. They are actually pretty useful and you will use them a lot when programming.

# Literal String Examples

```
 9        System.out.println("Float minimum value = " + myMinFloatValue);
10        System.out.println("Float maximum value = " +myMaxFloatValue);
11
12        double myMinDoubleValue = Double.MIN_VALUE;
13        double myMaxDoubleValue = Double.MAX_VALUE;
14        System.out.println("Double minimum value = " + myMinDoubleValue);
15        System.out.println("Double maximum value = " +myMaxDoubleValue);
```

# String Datatype

The **String** is a datatype in Java, which is not a primitive type.  It's actually a **Class**, but it enjoys a bit of favoritism in Java to make it easier to use than a regular class.

# What is a String?

```
5   public static void main(String[] args) {
6
7       char myChar = 'D';
8       char myUnicodeChar = '\u0044';
9       System.out.println(myChar);
10      System.out.println(myUnicodeChar);
11      char myCopyrightChar = '\u00A9';
12      System.out.println(myCopyrightChar);
13      boolean myTrueBooleanValue = true;
14      boolean myFalseBooleanValue = false;
15
16      boolean isCustomerOverTwentyOne = true;
17  }
18 }
```

A **String** is a sequence of characters. In the case of the **char** which you can see above which we discussed in the previous video, it could contain a **single character** only (regular character or Unicode character).

A **String** can contain a sequence of characters.  A large number of characters. Technically it's limited by memory or the **MAX_VALUE** of an **int** which was **2.14 Billion**. That's a lot of characters.

# Immutable Strings

When I said you can delete characters out of a **String**, that's not strictly true. Because **Strings** in Java are immutable. That means you can't change a **String** after it's created. Instead, what happens is a new **String** is created.

# Immutable Strings - 2

```
24          int myInt = 50;
25          lastString = lastString + myInt;
26          System.out.println("LastString is equal to " + lastString);
27          double doubleNumber = 120.47d;
28          lastString = lastString + doubleNumber;
29          System.out.println("LastString is equal to " + lastString);
30
```

So in the case of this code, **lastString** doesn't get appended the value "**120.47**" instead a new **String** is created which consists of the previous value of **lastString** plus a text representation of the double value **120.47**.

The net result is the same, **lastString** has the right values, however, a new **String** got created and the old one got discarded.

Don't worry if this makes no sense, it will later in the course. For now, I just wanted to point out that **Strings** are **immutable**.

# The Code We Used To Append Strings Was Inefficient

As a result of a **String** being created, appending values like this is inefficient and not recommended. I'll show you a better way of doing it in a future video where we discuss something called a **StringBuffer** which can be changed.

We need an understanding of classes before we tackle **StringBuffer**, and also, you will come across code written in the style I've used in this lecture so it's still useful for you to know how to do things this way.

# What are Operators?

**Operators** in Java are special symbols that perform specific operations on one, two, or three operands, and then return a result. As an example, we used the **+** (addition) operator to sum the value of two variables in a previous video.

```
27          double doubleNumber = 120.47d;
28          lastString = lastString + doubleNumber;
```

There are many others in Java.

# What is an Operand?

An **operand** is a term used to describe any object that is manipulated by an **operator**. If we consider this statement

**int myVar = 15 + 12;**

Then **+** is the operator, and **15** and **12** are the **operands**. Variables used instead of literals are also **operands**.

**double mySalary = hoursWorked * hourlyRate;**

In the above line **hoursWorked** and **hourlyRate** are **operands**, and ***** (multiplication) is the **operator**.

# More on Expressions

An **expression** is formed by combining variables, literals, method return values (which we haven't covered yet) and operators.

In the line below, **15 + 12** is the expression which has (or returns) **27** in this case.

**int myValue = 15 + 12;**

# More on Expressions - 2

In the statement below, **hoursWorked * hourlyRate** is the expression. If **hoursWorked** was **10.00** and **hourlyRate** was **20.00** then the expression would return **200.00**;

**double mySalary = hoursWorked * hourlyRate;**

Lets explore Operators in more detail.

# Comments in Java

**Comments** are ignored by the computer and are added to a program to help describe something. **Comments** are there for humans.

We use **//** in front of any code, or on a blank line.  Anything after the **//** right through to the end of the line is ignored by the computer.

Aside from describing something about a program, comments can be used to temporarily disable code.

# 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 test 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**.

Let's see how this works in practice.

# if-then Recommendation - Use Code Blocks

```
47          boolean isAlien = false;
48          if (isAlien == true)
49              System.out.println("It is not an alien!");
50              System.out.println("And I am scared of aliens");
51      }
52  }
```

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

A **code block** allows more than one statement to be executed - a block of code.

The format is:

```
if (expression)  {

  // put one or more statements here

}
```

# Conditions in Java

A **condition** is an expression that returns a **boolean** value.

# Conditional Statements

The **if-then** statement starting on **line 59** is one of a number of conditional statements in Java. As you have seen it's a way to execute code based on a condition.

We'll be seeing more conditional statements as we progress through the course, as well as the **if else** statement which is an enhanced form of the **if-then** statement we have used so far. More on that later.

# Logical AND and Logical OR

The **AND** operator comes in two flavors 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**.

# Logical AND and Logical OR - 2

You will almost always want to be doing this. The **&** is a bitwise operator working at the bit level. This is an advanced concept that we won't get into here.

Likewise **||** is the Logical **OR** what again operates on **boolean** operands - Checking if a given condition is **true** or **false**.

Again, you will almost always want to be doing this. The **|** is a bitwise operator working at the bit level.

# Assignment and Equal to Operators - Differences

```
67          int newValue = 50;
68          if (newValue = 50) {
69              System.out.println("This is an error");
70          }
```

As you can see in the code we typed to declare the **newValue int**, it's using the **assignment operator** (one equal sign) to assign the value **50** to **newValue**.

What we needed to do in the code is to not use the **assignment operator** in the **if-then** statement, but rather the **equal to** operator which has two equal signs.

# Assignment and Equal to Operators - Differences - 2

```
67          int newValue = 50;
68          if (newValue = 50) {
69              System.out.println("This is an error");
70          }
```

This is what the code should look like:

```
67          int newValue = 50;
68          if (newValue == 50) {
69              System.out.println("This is an error");
70          }
```

We are not assigning a value here, we want to test if the values are equal to each other.

# The NOT Operator

The ! or NOT Operator is also known as the Logical Complement Operator.

For use with **booleans** it tests the alternate value - we saw **(isCar)** tests for **true**, by adding a **!** operator before the value we can check the opposite - **false** in this case.

**boolean isCar = false;**

We can use either of these statements:

**if (isCar == false)**

**if (!isCar)**

to check if the **boolean isCar** is **false**.

# The NOT Operator - 2

We can use either of these statements:

**if (isCar == false)**

**if (!isCar)**

to check if the **boolean isCar** is **false**.

I'd generally recommend using the abbreviated form of both for two reasons. One to avoid the potential for error by accidentally using an assignment operator, and second, the code is more concise.

# Ternary Operator

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

It's a shortcut of the **if-then-else** statement (**else** we will be discussing later).

# Ternary Operator - 2

**int ageOfClient = 20;**

**boolean isEighteenOrOver = ageOfClient == 20 ? true : false;**

Operand one - **ageOfClient == 20** in this case is the condition we are checking. It needs to return **true** or **false**.

Operand two - **true** here is the value to assign to the variable **isEighteenOrOver** if the condition above is **true**

Operand three - **false** here is the value to assign to the variable i**sEighteenOrOver** if the condition above was **false**.

# Ternary Operator - 3

**int ageOfClient = 20;**

**boolean isEighteenOrOver = ageOfClient == 20 ? true : false;**

In this case, **isEighteenOrOver** is assigned the value **true** because **ageOfClient** has the value **20**.

It can be a good idea to use **parentheses** like this to make the code more readable.

**boolean isEighteenOrOver = (ageOfClient == 20) ? true : false;**

# Keywords and Naming Conventions

A keyword is one of around 57 words in the Java language that have a specific meaning. I say around because from time to time new keywords are added in a new version of Java.

You cannot use a keyword as a variable name. And the same rule applies to methods, class names, or other identifiers in Java. You can't use them to name any of those things. We will discuss methods, class names, and these other identifiers later in the course.

So keywords are reserved words, not to be used for any other purpose.

A list of Java Keywords is available at this link.

https://en.wikipedia.org/wiki/List_of_Java_keywords

# What is a Naming Convention?

Think of a naming convention as a generally agreed scheme for naming things.

If you follow a convention it means you are using a consistent approach to naming things.

And if other programmers do the same it makes the code easier to read because everyone is using a similar way to name things.

# Java Naming Conventions - 1

Java is a flexible language and allows you to use a range of naming conventions, or no convention for your variables, methods, class names, and so on.

Not following a convention is a bad idea though.

You should follow a convention and get into good habits from day 1 when learning Java (or any language for that matter).

Since we haven't really discussed methods and class names yet, let's discuss naming conventions for variables at this time in Java.

# Java Naming Conventions - 2

While you can use names like myInt or variable1 or even MYNUMBER6, they are not great names.

Java does have naming conventions.  And it's a good idea to adopt these conventions since employers will assume you know them and will use them.

I've got a video on Youtube which talks more about code conventions, check out the Youtube link in the resources section of this video for a discussion on code conventions in general which is useful for you to know.

# Variable Naming Conventions - 1

Use proper LowerCamelCase when naming variables in Java.

Ensure the variable name gives an idea on the variables purpose.

# Variable Naming Conventions - 2

Bad naming example.

int myInt = 50;

This does follow lowerCamelCase convention but doesn't signify what the variable is being used for. You might be clear on it's intent now, but imagine coming back to this code in 12 months.  Will you still remember what its purpose was?

And what if you are looking at someone else's code - that variable name gives no clue as to its purpose. It makes it harder for you to get up to speed with the code.

# Variable Naming Conventions - 3

Contrast that to a better example.

int studentsAge = 50;

We are still following lowerCamelCase convention but now we can see at a glance that the variable is used to record the student's age.

Make sure you get into the habit of doing this - habits good or bad are formed when learning something new, so take the time to learn a good habit!

We'll talk more about code conventions for methods, classes, etc when we start exploring them in more detail later in the course.

# Expressions in Detail

An expression is a construct that evaluates to a single value.

Expressions are constructed from operands and operators, which we have seen in the last section.

The operators in an expression indicate which operations to apply to the operands.

The order of evaluation of operators in an expression is determined by the precedence and associativity of the operators.

We talked about operator precedence in the last section. If you recall Precedence is the priority order of an operator - it's what order Java processes operators in an expression.

# What is Associativity?

Associativity is the direction of execution of operators and can be either left to right or right to left.

If consecutive operators in an expression have the same precedence, associativity is used to decide the order in which those operators are evaluated.

As mentioned, an operator can be left-associative, or right-associative. Technically an operator can be non-associative but we won't discuss that at this point.

# Java Operator Associativity - 1

Left-associative operators of the same precedence are evaluated in order from left to right. For example, addition and subtraction have the same precedence and they are left-associative.

In the expression 16 **-** 8 **+** 5, the subtraction is done first because it is to the left of the addition operator, which is left-associative. This is what left-associative effectively means here.

# Java Operator Associativity - 2

Right-associative operators of the same precedence are evaluated in order from right to left. For example, assignment is right-associative. Consider the following code fragment:

```
int numberTrucks = 5;
int numberCars = 7;
int fleetCount = numberTrucks + numberCars;
```

The expression to the right of the equals sign must be evaluated first in order to know what value to store in the variable.

So equals or assignment is right associative.

# Statements

Statements can be compared to sentences in natural languages. A statement forms a complete unit of execution.

There are three types of Statements in Java as mentioned.

# The Declaration Statement

The "Declaration" part means the statement declares a variable, and we've seen numerous examples so far in the course of a declaration statement. We just didn't know what it was called.

```
int finalScore = 50;
```

The data type at the front and semi-colon at the end make it a declaration statement in Java.

# The Expression Statement - 1

As the name suggests expression statements have a lot to do with expressions. Some but not all expressions can become expression statements in Java by adding a semi-colon to the end of the line.

The four expression groups that can become expression statements are...

# The Expression Statement - 2

a) Assignment expressions

```
groupAge = 5
salary *= 2
profitMade += 10
```

Adding semi-colons to these expressions convert them into expression statements.

# The Expression Statement - 3

a) Assignment expressions

```
groupAge = 5;
salary *= 2;
profitMade += 10;
```

Adding semi-colons to these expressions convert them into expression statements.

# The Expression Statement - 4

b) Any use of the increment operator ++ or the decrement operator --

```
numberStudents++
surveyResults--
```

A semi-colon at the end of the line converts these into expression statements.

# The Expression Statement - 5

b) Any use of the increment operator ++ or the decrement operator --

```
numberStudents++;
surveyResults--;
```

A semi-colon at the end of the line converts these into expression statements.

# The Expression Statement - 6

c) Method calls

```
System.out.println("Testing");
```

d) Object creation expressions.

```
Car ownersCar = new Car();
```

I'm putting c and d here for completeness, but we haven't discussed those much or at all, but will later in the course.

So that's the four types of expressions that become expression statements by adding a semi-colon to the end of the line.

# The Control Flow Statement - 1

The third type of statement in Java is the Control-Flow Statement.

The statements inside your source files are generally executed from top to bottom, in the order that they appear.

Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code.

# The Control Flow Statement - 2

Control Flow statements also include the if-then-else, looping statements (for, while, and do-while) and then branching statements (break, continue, and return). We will cover all of these in future videos.

# Increment and Decrement Operators

When we add a **--** or a **++** to a variable it decrements or increments the value by 1. We've seen that before.

But we can dictate when the decrement or increment happens, either before or after the expression is evaluated.

This is referred to as Prefix and Postfix.

# How Prefix and PostFix works in Java - 1

Lets review the code below.

```
int playerLives = 3;

System.out.println("Lives = " + playerLives--);  // Outputs 3
```

This returns the value 3, because playerLives was read and the value obtained 3, before the decrement was executed.  The decrement was postfix, done after the output happened.

If we added another line of code directly below

```
System.out.println("Lives = " + playerLive); // Outputs 2
```

It will show as 2, because playerLives was decremented by 1.

Let's look at pre-fix.

# How Prefix and PostFix works in Java - 2

In this case

```
int playerLives = 3;

System.out.println("Lives = " + --playerLives);  // Outputs 2
```

This returns the value 2, because playerLives was read and decremented by 1, and the result returned.  The decrement was prefix, done before the value was outputted.

# White Space - 1

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

Java treats code like this:

```
int planesDestroyed = 50;planesDestroyed--;System.out.println("Planes destroyed = " + planesDestroyed);
```

The same as code like this:

```
int planesDestroyed = 50;
planesDestroyed--;
System.out.println("Planes destroyed = " + planesDestroyed);
```

Obviously, in the second case, it's a lot easier to read for us mere mortals.  Which is why the use of whitespace is recommended.

# White Space - 2

White space does not take up any memory in your final program, and that's because Java reads your java code, strips out all the extra spaces, before converting into an executable program.

It does take up more memory in your source file i.e. the file you edit in IntelliJ or any editor.

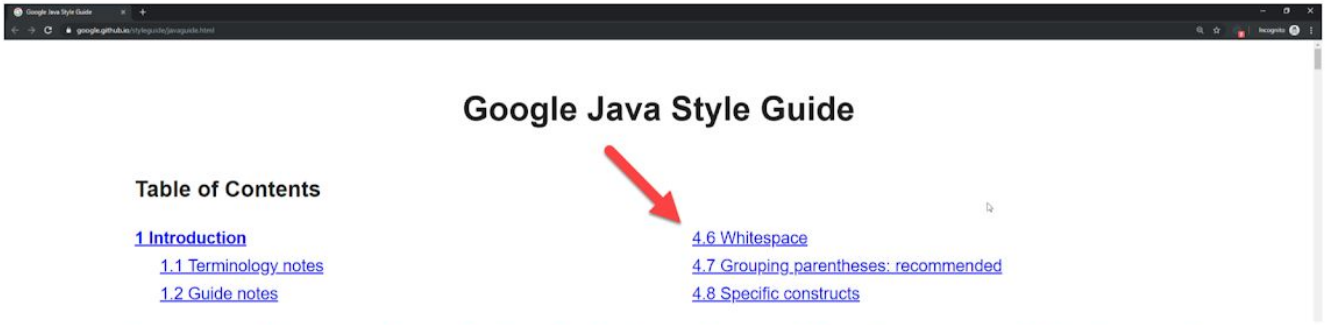
But it has no impact on the final program that is run, the execution speed of that program, or the size of that program.

Use whitespace freely, but don't go overboard - it is possible to use too little or too much whitespace which can have a negative impact on the human readability of your code.

# White Space - 3

Code conventions for White space exist which you can refer to for more detail.

The google java style guide we have seen previously in this course has a section on whitespace, so refer to that for more information. The link is again in the resources section of this video.




https://google.github.io/styleguide/javaguide.html

# Code Block

A code block in Java is one or more statements within left and right curly braces. All the code within is part of a logical unit.

When we define a class like this:

```
1  public class Main {
2
3  }
```

That's a code block.

# Code Block

Code blocks can nested within other code blocks.

```
1  public class Main {
2      public static void main(String[] args) {
3
4          int salary = 60;
5      }
6  }
```

# Code Block

And even:

```
1  public class Main {
2      public static void main(String[] args) {
3          int salary = 60;
4          if (salary > 50) {
5              System.out.println("You are rich!");
6          }
7      }
8  }
```

This simplistic little program has three code blocks, the first is here.

The second code block is here, and the third block is here.

# Code Block

```
1  public class Main {
2      public static void main(String[] args) {
3          int salary = 60;
4          if (salary > 50) {
5              System.out.println("You are rich!");
6          }
7      }
8  }
```

So code block 1, this one, contains code block 2 and code block 3. Code block 2 contains code block 3.

We say that code block 3 is a nested block of code block 2 and code block 2 and code block 3 and nested code blocks of code block 1.

# Code Block

```
1  public class Main {
2      public static void main(String[] args) {
3          int salary = 60;
4          if (salary > 50) {
5              System.out.println("You are rich!");
6          }
7      }
8  }
```

This will become important later when we discuss variable scope, but for now, keep in mind that these are logical units.

I mentioned indentation, so let's discuss that and then see what it has to do with code blocks.

# Indentation

In computer programming, indentation is a convention of adding spaces to typed code to convey program structure.

Here is our code blocks sample code again.

```
1  public class Main {
2      public static void main(String[] args) {
3          int salary = 60;
4          if (salary > 50) {
5              System.out.println("You are rich!");
6          }
7      }
8  }
```

# Indentation

```
1  public class Main {
2      public static void main(String[] args) {
3          int salary = 60;
4          if (salary > 50) {
5              System.out.println("You are rich!");
6          }
7      }
8  }
```

Notice how there are spaces to the left of the word public, in the main method declaration.

Note also that there are spaces to the left of the if statements, and spaces to the left of the System.out.println statement.

This is indentation.

# Indentation

```
1  public class Main {
2  public static void main(String[] args) {
3  int salary = 60;
4  if (salary > 50) {
5  System.out.println("You are rich!");
6  }
7  }
8  }
```

If we contrast that to the same code, without any indentation.

This code works identically to the indented code - Java ignores indentation when it scans your program, so just like whitespace, indentation is for humans only - it makes code easier to read and for you to see at a glance which code blocks code is part of.

# Indentation

```java
public class Main {
    public static void main(String[] args) {
        int salary = 60;
        if (salary > 50) {
            System.out.println("You are rich!");
        }
    }
}
```

```java
public class Main {
public static void main(String[] args) {
int salary = 60;
if (salary > 50) {
System.out.println("You are rich!");
}
}
}
```

Looking at both samples of code at the same time, to the left we have the indented code, and to the right the code that is not indented.  One is a lot easier to read, right?

# Indentation

```
1  public class Main {
2      public static void main(String[] args) {
3          int salary = 60;
4          if (salary > 50) {
5              System.out.println("You are rich!");
6          }
7      }
8  }
```

# Indentation

```java
public class Main {
    public static void main(String[] args) {
        int salary = 60;
        if (salary > 50) {
            System.out.println("You are rich!");
        }
    }
}
```

```java
public class Main {
public static void main(String[] args) {
int salary = 60;
if (salary > 50) {
System.out.println("You are rich!");
}
}
}
```

Again, Java does not care whether you indent your code or not, but it makes sense to do so, and there are code conventions about indentation.

# Code Block Recap

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

You can use a code block wherever a single statement is allowed.

The format of a code block is: An open curly brace **"{"**, followed by one or more statements, followed by a closing curly brace **"}"**.

There are places in Java where code blocks are mandatory - as we saw when declaring the **main()** method. Likewise, there are places where code blocks are optional.

# Code Block Recap

Within an if-then statement, use of a code block, although optional, should always be used. As follows:

```
1    if (expression) {
2
3            // put one or more statements here
4
5            }
```

# Naming Convention Note

As mentioned in an earlier video, when a Boolean variable is declared, good naming convention is to start the variable name with "**is**".

Another rule of thumb is to have the portion of the Boolean variable name, following "**is**" state how the variable will be used.

Also choose the name to be positive.  Consider this:

```
1        boolean isValid = true;
```

is preferred over:

```
1        boolean isInvalid = false;
```

# Naming Convention Note

Also choose the name to be positive.  Consider this:

```
1        boolean isValid = true;
```

is preferred over:

```
1        boolean isInvalid = false;
```

The reason for this is you want to strive to make your code as readable as possible.

Continuing with isValid vs. isInvalid: If you saw in code you were reviewing "**!isInvalid**" (NOT isInvalid).  Requires the reader to stop and decipher the true intention.

# Scope

"**Scope**" refers to when a variable can be accessed.  (Actually Scope is another subject which requires video in it's own right).

Java can access variables from outer code blocks (which are still executing).

Java cannot access variables inside code blocks which have finished executing.

Java cannot access variables inside code blocks which have not started executing.

# Methods Overview

Methods are a way to organize your code to reduce duplication and make your code easier to maintain.

We have seen methods in use a lot in the course to date, but haven't talked much about them.

Methods can be placed in any order in the file.

The general format of defining a method in Java is:

<MODIFIER> <RETURN  TYPE> methodName (<PARAMETERS>) {

}

# Methods Overview

A sample method declaration is as follows:

```
public static void main(String[] args) {

}
```

Which of course is the main method we have been using so far in the course so far.

Here we are using two modifiers public and static and a return type void, meaning the method doesn't return anything.

We are still not going to discuss modifiers or return types yet, but that is coming later in the course.

# Methods Overview

```
public static void main(String[] args) {

}
```

A method name is always followed by an open curly brace and always contains a closing curly brace.

When creating a method, a good programming practice is to choose a method name using one or more words that describe an action.

# Method Parameters Overview

One use for methods is to make calculations where the same calculation is needed repeatedly throughout out our code.

It would make our methods more useful if we can pass different values to be used in making those calculations.

Defining parameters for our methods allow us to do just that.

To recap, the general format of defining a method in Java is:

<MODIFIER> <RETURN  TYPE> methodName (<PARAMETERS>) {

}

# Method Parameters Overview

Each method can have zero or more PARAMETERS.

A single parameter is a data type and variable name.

Multiple parameters are separated by a comma.

# Method Parameters Overview

Here is a sample declaration of a method that uses four parameters:

```
public static void calculateScore(boolean isGameOver, int score, int levelCompleted, int bonus) {

}
```

Here is how you would declare a method that has no parameters:

```
public static void doSomethingUseful() {

}
```

Notice that the PARAMETERS are always surrounded with "(" and ")". Even if there no parameters the "(" and ")" are still required.

# Method Parameters Overview

Well, we would normally have to create a second method to do this second part of the code.

But of course, doing that would mean that we are effectively duplicating our work again. So we haven't really fixed anything.

All we've really done would have been just to convert our two code blocks to two methods.

So there must be a better way of doing that.

Well, there actually is with methods, and that is using the concept of parameters.

# Method Parameters Overview

What would be ideal is to have one method and pass the information to it each time we used it - i.e. **isGameOver**, **score**, **levelCompleted** and **bonus**.

That way, the **calculateScore()** method doesn't have to create these variables, they are "sent" to the method ready for use.

# Method Parameters Overview

Well, as it turns out, we can do that. We can actually pass information to a method.

So what we need to do is we need to look in this bracket area **calculateScore()** and we need to define the parameters that are part of this.

So, what we need to do is define in the method what parameters or what information we're sending through to it from our main method, from where we're actually calling the method.

# Returning Values from Methods

Sometimes it's useful to have a method return a value. One use is when the result of a method is used in several places.

It would be ideal to perform the calculation once, store the result in a variable, and be able to use the variable as many times as needed without having to perform the calculation each time.

Fortunately making a method return a value is simple.

# Returning Values from Methods

As we saw in the two previous videos, the general format of defining a method in Java is:

<MODIFIER> <RETURN_TYPE> methodName (<PARAMETERS>) {

}

So far we have only used methods with a **<RETURN_TYPE>** of void, meaning nothing is returned.

# Returning Values from Methods

<MODIFIER> <RETURN_TYPE> methodName (<PARAMETERS>) {}

To have your method return a value you specify the data type of the information being returned.

- Methods can only return a single value.
- If the **<RETURN_TYPE>** is something other than void the last statement executed MUST be return followed by the value to be returned.

# Methods Recap

Methods are a way to organize your code to reduce duplication and make your code easier to maintain.

Methods:

- Can be placed in any order.
- Can optionally be passed data by using parameters.
- Declared parameters for a method effectively become variables created automatically that are local to the method and are no longer accessible when the method stops executing.

# Methods Recap

- Can optionally return a single value - later in this course, I'll discuss returning Objects that contain multiple values.

Methods that do not return values are sometimes called **PROCEDURES**.

Methods that do return a value are sometimes called **FUNCTIONS**.

*The terms: method, procedure and function are interchangeable.*

# Methods Recap (continued...)

Method declarations contain the following:

1. For right now all our methods begin with the words: public static - these are access modifiers, and we will learn more about these later in the course.
2. Return type:
    - Use *void* if not returning a value.
    - If returning a value the *data type* of the return value must be specified.

# Methods Recap (continued...)

3. Name:
    - Should begin with a lower case letter. Use capital letters at the start of each new word used in the method name.
    - Uses one or more words that describe an *action*.

# Methods Recap (continued...)

4. Parameter list:
    - All method names are immediately followed by left and right parenthesis.
    - If parameters are used they are added inside the parenthesis via a comma separated list of data types and variable names.
    - If no parameters are defined for a method the parenthesis are still required.

# Methods Recap (continued...)

Examples:

A. public static int calculateScore(boolean isGameOver, int score, int levelCompleted, int bonus)

- Returns an int value has 4 parameters.

B. public static void doSomething()

- Does not return a value, uses no parameters.

# Challenge Slides.

# Challenge

```
1  public class Hello {
2
3      public static void main(String[] args) {
4          System.out.println("Hello World");
5      }
6  }
7
```

How would you go about changing this to print my name, instead of World?

Looking at this code see if you can figure it out.

# Challenge

Hello.java

```
1  public class Hello {
2
3      public static void main(String[] args) {
4          System.out.println("Hello, Tim");
5
6          int myFirstNumber = 5;
7      }
8  }
```

Looking at the first **System.out.println** statement, what do you think should be typed on the line below our declaration statement to print out the value of the variable **myFirstNumber**?

# Challenge

**Create additional variables**.

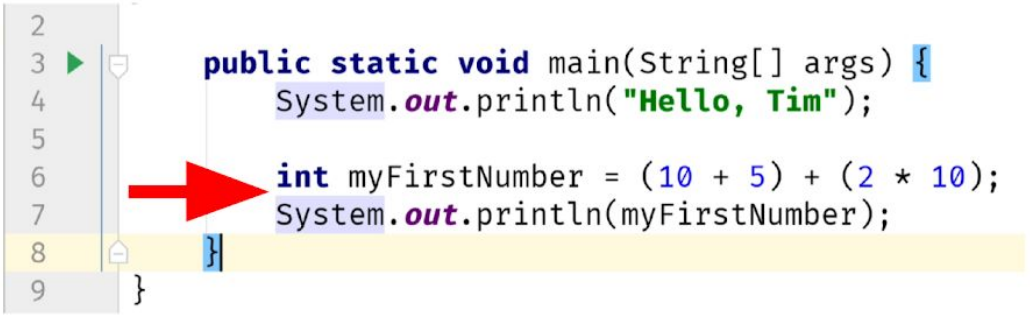
Declare the following variables and add to our program.

**mySecondNumber** which is an **int** and assign a value of **12** to it.

**myThirdNumber**, also an **int** with a value of **6**.

# Challenge

Put the declaration statements above the **System.out.println** - you'll see why later.



# Final Variable Challenge

Create a new variable call **myLastOne**.

We want the value to be **1000 less** the current value of **myTotal** - the data type is an **int**.

Print out the value of **myLastOne**.

Hint:  Use another operator that we haven't actually used before, but should be easy to figure out if you think about it for a bit.

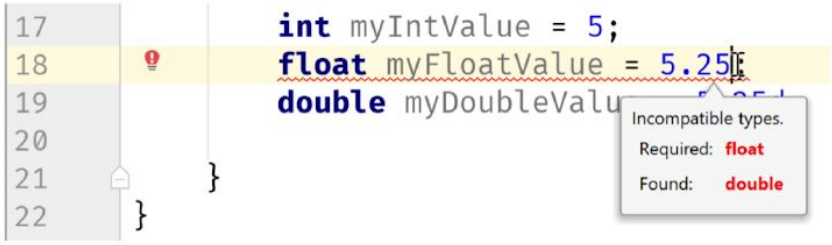
# Primitive Types Challenge

Your challenge is to create a **byte** variable and set it to any valid **byte** number, it doesn't matter. Create a **short** variable and set it to any valid **short** number.

Create an **int** variable and set it to any valid **int** number. Lastly, create a variable of type **long** and make it **equal to 50000 plus 10 times** the **sum** of the **byte plus** the **short plus** the **integer** values.

# Challenge

Thinking back to our discussion on casting which we used to convert both a **byte** and a **short** to the **int** equivalent, how do you think you would do the same for the **float** to remove the error?



# Challenge

Convert a given number of pounds to kilograms.

STEPS:

1. Create a variable with the appropriate type to store the number of pounds to be converted to kilograms.
2. Calculate the result i.e. the number of kilograms based on the contents of the variable above and store the result in a 2nd appropriate variable.
3. Print out the result.

HINT: **1** pound is equal to **0.45359237** of a kilogram. This should help you perform the calculation.

# Operator Challenge

Let's test what you have learned about **Operators**.

1. Create a **double** variable with a value of **20.00**.

2. Create a second variable of type **double** with the value **80.00**.

3. Add both numbers together and multiply by **100.00**.

4. Use the **remainder** operator to figure out what the **remainder** from the result of the operation in **step 3** and **40.00**. We used the **modulus** or **remainder** operator on **int's** in the course, but we can also use it on a **double**.

5. Create a **boolean** variable that assigns the value **true** if the remainder in **#4** is **0**, or **false** if its **not zero**.

6. Output the **boolean** variable.

7. Write an **if-then** statement that displays a message **"Got some remainder"** if the **boolean** in **step 5** is **not true**. Don't be surprised if you see output for this step, where you might expect it not to show. I'll explain it in my solution.

# Challenge

Armed with this knowledge, your challenge is as follows.

1. Create a variable of type double to track the number of miles driven in a month by a companies sales rep.
2. Create a variable of type int to track the number of customers who visited a store in a day.
3. Create a variable to record the last key pressed in a game.

Use the proper naming conventions discussed in the previous video for your variable names.

# Mini Challenge

So a quick challenge for you, how do you think we would change the code to **return a -1 if isGameOver was equal to false?**

# Challenge

The challenge involves creating two methods.

The first one should be called calculateHighScorePosition.

It should have one parameter, score which is an int.

The method should return an value that is calculated based on the score.

return 1 if the score is >= 1000

return 2 if the score is >= 500 and < 1000

return 3 if the score is >= 100 and < 500

return 4 in all other cases

# Challenge

The second method should be called displayHighScorePosition.

It should have two parameters, name and position.

Name is the players name and a String.

Position is the value returned from the first method and represents where in the high score table the players score belongs.

# Challenge

This method does not return anything, rather, it outputs information about the player and the position.

Output should be:

<Name> managed to get to position <Position> on the high score table.

<Name> and <Position> would be replaced with the actual name and position passed to the method.

If the arguments passed to displayHighScorePosition were **Tim** and **2**, then the output should be

Tim managed to get to position 2 on the high score table.

# Challenge

Call calculateHighScorePosition four times, passing the score 1500, 900, 400, and 50.

Save the value returned from the method and pass it along with a player name to displayHighScorePosition.