Getting Started

In this video, we're going to print some things to the screen, to get our first real taste of programming.

The Java Shell Tool, or JShell as it's referred to, is the tool we'll be using to achieve this, and is the one I talked about in the previous video.

Be sure to watch that previous video if you need a recap, on how to access it for your operating system.

For best results for this course, make sure you follow along as closely as you can, typing the code segments I'll be typing, into your own JShell session.

Getting Started

Nothing beats learning from your own mistakes, and figuring out how to get a line of code to work, when it wasn't working before.

This emulates real-world problem solving, and you'll find when you type in your own code, you'll be making all kinds of typos, or omitting or overlooking some characters that are required.

This is the best way to learn, and pick up all the nuances, of any programming language you're learning.

If you do encounter a mistake on your own, pause the video, try solving it on your own, and if that doesn't work – remember the working code segments, are always available in the resources section of every video.

Getting Started

In addition, I'll be trying to talk through, and demonstrate, some of the most common mistakes we all make in Java. If you're making these mistakes, and correcting them on your own, you won't forget them easily.

If you're like most programmers, figuring out how the pieces fit, and seeing the result of the code working, after a bit of a struggle, is the best part of programming.

Coding is a lot of fun, and I promise you'll feel the same way as we progress through this course.

Getting Started

All right – So let's get started. It's always been a tradition, when you're learning a new programming language, to create a simple program that outputs the text, "Hello World". This lets us test all the parts that are necessary, to create a successful program, with very little investment of time, or knowledge of the language.

It's the first step, and honestly, even after many years of programming, I still get that first feeling of achievement, when I've accomplished this step in a new language.

If you can print "Hello World" to the screen, you're on your way to bigger, and better things.

Lets do that now.

Statement

What is a statement?

It's a complete command to be executed. It can include one or more expressions, and I'll be talking about expressions and related topics, as we progress through the course.

Statement

```
System.out.print("Hello World");
```

What we've typed above, is a command to print some information to the screen, using syntax provided by the Java language.

We specified what we wanted Java to print in the parentheses and double quotes – in this case, the text "Hello World" – effectively we're telling Java to print out the words, as we've specified them in the quotes – "Hello World".

After we press the **enter** key, the program executing the code should print that text to the screen – in this case, the output is printed on the very next line.

Challenge

I've got a bit of a challenge for you here, and the challenge is to see whether you can modify the statement we typed below, so that instead of it printing out "Hello World", it prints, "Hello Tim".

```
System.out.print("Hello World");
```

Looking at this code, how do you think you'd go about achieving that?

Let's see if you can figure that out, and get that to run.

Challenge

```
System.out.print("Hello World");
```

You have a few options here. You could just retype the entire line in from scratch, with the changed text to print out. But we want to modify the statement, not type a new line in.

To modify the existing line, press the up arrow key – remember that in JShell, you can see the history of the lines you've previously typed, using the up and down arrows.

Pressing the up arrow now, will display the code we first typed in – so make the change, and then press enter to see the result printed to the screen.

Hello World

Congratulations if you managed to make that work! These are the first steps almost all programmers make, in almost every programming language, regardless of their experience.

If you had a problem getting it to work, don't worry, the more you work at this course, the easier things will be to understand – so hang in there!

Every video builds on some content in previous videos, and you'll have lots of opportunities to explore these concepts further, so you won't be left behind.

Next, let's look at some common errors you might get, when typing in Java code.

Hello World

These were a few examples, of errors that might occur, when your coding in JShell. Let me encourage you to play with this line of code in JShell, in as many ways as you can think of, to see what kind of errors you might get, or what kind of output is produced.

This is the whole point of JShell – to provide you with a safe place to test code segments.

Remember that the key combination 'control c', on windows, or 'control d' on a mac or a linux machine, should cancel what you are in the middle of, and get you back to the JShell prompt.

Hello World

Typing forward slash and the word '**exit**', or forward slash with the shortcut text **ex**, will end your JShell session, if you get stuck.

An example would be **/exit** or **/ex**.

Variables

In the previous video, we saw how to write our first code in Java. It was pretty basic to say the least, printing out some text on the screen, but it was a good place to start.

In this video, I want to talk about variables and keywords.

Keywords

A keyword is **any one of a number of reserved words, that have a predefined meaning in the Java language**.

In Java syntax, all code is case-sensitive, and this includes keywords. As we'll soon see, an **int**, all in lowercase, is not the same as **Int**, with a capital I. Here, an **int**, (all in lowercase) is a keyword in Java.

Variables

So what are variables?

Well, **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 computer's **random access memory**, or RAM.

Data type

There are lots of different types of data, that we can define for our variables, some of which I've shown you in the keyword list previously.

Collectively, these are known as data types.

As you may have guessed, some data types are keywords in Java. When we get to the Object Oriented features of Java, you will see that we have a lot of flexibility for creating our own data types, but in the next couple of videos, we'll explore primitive data types, which are built into the Java language.

Declaration Statement

A declaration statement is used to define a variable by indicating the data type, and the name, then optionally to set the variable to a specific value.

Expressions

So what is an expression?

An expression is a coding **construct, that evaluates to a single value**.

Variable Challenge 1

```
jshell> System.out.print("Hello World");
Hello World
```

```
C:\Users\LPA>jshell
|  Welcome to JShell -- Version 17.0.2
|  For an introduction type: /help intro

jshell> int myFirstNumber = 5;
myFirstNumber ==> 5

jshell>
```

Your challenge is to look at creating a new System.out.print below the declaration for **myFirstNumber**, and to see if you can figure out how to print out the value of the **myFirstNumber** variable.

Variable Challenge 2

Here's your challenge – Change the value of our existing variable from ten to one thousand, and print the new value out using the System.out.print method as we've done previously.

Variable Declarations in JShell

By declaring a variable again, we are effectively re-declaring a variable, and in normal Java programming, that would not be allowed, and would throw an error.

Due to its interactive nature, JShell “holds our hand”, and allows the redeclaration to occur without throwing an error.

For now, just follow along, knowing that re-declaring a variable for a second, or subsequent time, is not allowed, and later in the course, when we switch to a full editor, you'll see what happens when we try and do that.

Note that we can assign a value to a variable multiple times in Java, but it’s the declaration (which includes the data type) that cannot normally be done a second time for the same variable.

Operators

Java operators, or just operators, perform an operation (hence the term) on a variable or value.

Addition, Subtraction, Division, and Multiplication are four common ones that I feel sure you're familiar with, but there are lots more operators you will work with as we go through the course.

Starting out with Expressions

In this video, we'll be continuing what we were working on in the last video, but here, we'll be using variables in expressions.

Remember the expression is the code segment that is on the right side of the equals sign in an assignment or declaration statement.

This code can be a simple literal value, like the **number 5**, or it can be a complex mathematical equation using multiple literal values and mathematical operators.

In this video, we'll talk about how to use variables to replace literal values.

Review

```
int myFirstNumber = (10 + 5) + (2 * 10);
```

Up until now, we've only used literal values in our expressions, and we've also used several operators, such as in the example we used in the last video – see above.

Challenge

Your challenge is to create two additional variables in JShell.

Here's what we need:

- One variable called **mySecondNumber**, which is an int, with a value of **12**.
- And another variable called **myThirdNumber**, also of type int, with a value of **6**.

Challenge

First, create a new variable and call it **myLastOne**:

- Its data type should be **int**.
- It should be set to the value of **1000**, minus (or less than) the value in the **myTotal** variable, which we've just talked about in our previous code segment.

Next, print out the value of the **myLastOne** variable on the line after you declare it.

Hint: We need to use another operator that we haven't used in code before, but if you think about this, it should be easy to figure out which operator you need to use.

Java code is case sensitive

Java code is case sensitive.

This includes not only keywords and language syntax, but variable names and data types as well.

myLastOne is not the same variable as **MyLastOne** with a capital **M**.

int in lowercase, is not the same as **Int** with the first letter capitalized, or **INT**, all in uppercase, etc.

Java code is case sensitive

Keywords need to be in lowercase.

And variables will always be exactly as you declare them, including capitalization.

Remember that case matters in Java code!

The **/vars** command in JShell can help you identify any misspellings you may have made.

Recap

In this video:

We used expressions to assign values to our variables, and we used variables we created, in expressions.

For now though, in our expressions, we've looked at only one data type so far, which is the type, int.

In the next video, we'll be looking at and discussing some additional data types.

Introduction

We've been working only with the **int** data type so far in the course.

In this video, we'll continue to look at **int**, as well as several other primitive types.

We'll also introduce the wrapper class, a special category of data type, which offers additional functionality that primitive types don't.

Java's Primitive Types

In Java, primitive types are the most basic data types.

The eight primitive data types in Java are shown in the table below, listed by the type of data stored for each:

Whole number	**Real number (floating point or decimal)**
byte short int long	float double
Single character	**Boolean value**
char	boolean

Java's Primitive Types

Consider these types as the building blocks of data manipulation.

Remember that primitive data types are simply placeholders in memory for a value.

What actually is an integer?

An integer is a whole number, meaning it doesn't contain a fractional element, or a decimal.

What values can we store in an integer?

There's a specified range of values allowed for the **int**, which is true for most data types.

What this means is, that the allowable range of values is NOT infinite.

There's a defined minimum, and maximum value, for each numeric data type, meaning you can't assign a number bigger or smaller (outside of that range).

Using the + sign in System.out.print

The plus sign, +, when used in **System.out.print** will print different data types together as a single line of text.

In the example:

```
System.out.print("Integer Minimum Value = " + myMinIntValue);
```

We want to print a label, before a numeric integer value.

Whatever follows the plus sign in **System.out.print** here, is converted to a **String** by Java, and concatenated to the **String** before it.

This is perfectly valid syntax in Java.

Classes

So what is a class?

A class is a building block for object-oriented programming, and allows us to build custom data types. We'll be talking more about classes in future videos.

Wrapper Classes

Java uses the concept of a wrapper class, for all of its eight primitive data types.

A wrapper class provides simple operations, as well as some basic information about the primitive data type, which cannot be stored on the primitive itself.

We saw that MIN_VALUE, and MAX_VALUE, are elements of this basic information, for the int data type.

Wrapper Classes

The primitive types, and their respective wrapper classes, are shown in the table below.

Primitive	Wrapper Class
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

You can see there, that in general, it's pretty easy to remember the wrapper class name, for your primitive data type. It's the same name, but with an uppercase letter at the start.

Wrapper Classes

Primitive	Wrapper Class
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

The wrapper classes for char and int, Character and Integer respectively, are the only two that differ in name (other than that first capitalized letter) from their primitive types.

The Integer Wrapper Class

In the code we just reviewed, we were able to use MIN_VALUE, and MAX_VALUE, on the wrapper class Integer.

```
int myMinIntValue = Integer.MIN_VALUE;
int myMaxIntValue = Integer.MAX_VALUE;
```

To discover the minimum and maximum range of numbers, that can be stored in an int, as we saw when we printed out these values previously:

```
Integer Value Range (-2147483648 to 2147483647)
```

Overflow and Underflow in Java

If you try and put a value larger than the maximum value into an int, you'll create something called an Overflow situation.

And similarly, if you try to put a value smaller than the minimum value into an int, you cause an Underflow to occur.

These situations are also known as integer wraparounds.

Overflow and Underflow in Java

The maximum value, when it overflows, wraps around to the minimum value, and just continues processing without an error.

The minimum value, when it underflows, wraps around to the maximum value, and continues processing.

This is not usually behavior you really want, and as a developer, you need to be aware that this can happen, and choose the appropriate data type.

When will you get an overflow? When will you get an error?

An integer wraparound event, either an overflow or underflow, can occur in Java when you are using expressions that are not a simple literal value.

The Java compiler doesn't attempt to evaluate the expression to determine its value, so it DOES NOT give you an error.

When will you get an overflow? When will you get an error?

Here are two more examples that will compile, and result in an overflow. The second example may be surprising. Even though we are using numeric literals in the expression, the compiler still won't try to evaluate this expression, and the code will compile, resulting in an overflow condition.

```
int willThisCompile = (Integer.MAX_VALUE + 1);
int willThisCompile = (2147483647 + 1);
```

If you assign a numeric literal value to a data type that is outside of the range, the compiler DOES give you an error. We looked at a similar example previously.

```
int willThisCompile = 2147483648;
```

What does an underscore mean in a numeric literal?

In Java, you cannot put commas in a numeric literal.

For example, the following is not valid syntax.

```
int myMaxIntTest = 2,147,483,647;
```

So Java provides an alternative way to improve readability, the underscore.

```
int myMaxIntTest = 2_147_483_647;
```

You can put the underscore anywhere you might want a comma, but you can't use an underscore at the start or end of the numeric literal.

Introduction

In the previous video, we saw that Java has eight primitive data types, and that the wrapper classes give us extra options.

Of these primitive types, half are used to store whole numbers (numbers without a fractional or decimal component), one of which we've explored already, the integer, or the int data type.

In this video, we'll take a look at the other three whole-number primitive data types.

byte, short, int, long

We've previously said that Java has four primitive data types used to store whole numbers, these are the byte, the short, the int, and the long.

Whole number Data Type	Wrapper Class	What's noteworthy
byte	Byte	Has the smallest range
short	Short	
int	Integer	Java's default data type for whole numbers
long	Long	Has the largest range

They are listed here in this table, by the range of values the type will support, the byte supports the smallest range, and the long supports the largest range.

The byte data type

The minimum value of a byte is -128.

The maximum value of a byte is 127.

Given its small range, you probably won't be using the byte data type much.

The byte wrapper class is the Byte with a capital B.

The short data type

The minimum value of a short is -32768.

The maximum value of a short is 32767.

The short wrapper class is the Short with a capital S.

byte and short overflow/underflow

Both the byte and the short, have the same overflow and underflow issue as the int data type has, but obviously with their own range of numbers.

Size of Primitive Types and Width

Size, or Width, is the amount of space that determines (or limits) the range of values we've been discussing:

Data Type	Width (in bits)	Min Value	Max Value
byte	8	-128	127
short	16	-32768	32767
int	32	-2147483648	2147483647

A byte, can store 256 numbers and occupies eight bits, and 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.

Using a numeric literal character suffix

The number 100, by default, is an int.

Java allows certain numeric literals to have a suffix appended to the value, to force it to be a different data type from the default type.

The long is one of these types and it's suffix is an 'L'.

This is one of the few instances Java is not case sensitive, a lowercase 'l' or an uppercase 'L' at the end of a whole number mean the same thing – the number is a long.

How big is the difference between an int and a long?

How big is the difference, in the range of values that a long can store, compared to the int?

You can see, from this table, that the difference is quite significant.

Data Type	Width (in bits)	Min Value	Max Value
int	32	-2147483648	2147483647
long	64	-9223372036854775808	9223372036854775807

When is L required?

A numeric literal that exceeds Integer.MAX_VALUE must use the 'L' suffix.

We cannot create a numeric literal in Java, that exceeds Integer.MAX_VALUE, without using the 'L' suffix, we'll always get the error 'integer number too large'.

Introduction

In the last video, I introduced you to three new data types. The byte, the short, and the long.

In this video, we'll be using these additional types in some basic arithmetic.

We've already done some math, using integers, but now, we'll be using these other data types.

We'll finish the video with a discussion about casting, which is a way to get Java to treat a variable of one type like a different data type.

I'll talk about when and why casting is sometimes necessary in Java code.

Rules for declaring multiple variables in one statem

- You cannot declare variables with different data types in a single statement.
- If you declare multiple variables of the same data type in a single statement, you must specify the data type only once before any variable names.

Assigning expressions to variables with data types that don't mat

The Java compiler does not attempt to evaluate the value, in a variable, when it's used in a calculation, so it doesn't know if the value fits, and throws an error.

```
byte myNewByteValue = (myMinByteValue / 2);
```

If your calculation uses literal values, Java can figure out the end result at compile time, and whether it fits into the variable, and won't throw an error if it does.

```
byte myNewByteValue = (-128 / 2);
```

In both examples, an int result is being returned from the calculation, but in the second example, Java knows the returned value can fit into a byte.

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 parentheses like this:

```
(byte) (myMinByteValue / 2);
```

Other languages have casting too, this is common practice and not just a Java thing.

What does it mean when Java defaults the data type to an

So, what effect does int, being the default value, have on our code?

Looking at the scenarios we just looked at in summary, we know the following:

This statement works because the result is an int, and assigning it to an int variable is fine

```
int myTotal = (myMinIntValue / 2);
```

This statement doesn't work, because the expression (myMinShortValue / 2) is an int, and an int can't be assigned to a short, because the compiler won't guess the

```
short myNewShortValue = (myMinShortValue / 2);
```

What does it mean when Java defaults the data type to an

This statement works, because the result of (-128 / 2) is an int, but when calculations use only literal values, the compiler can determine the result immediately, and knows the value fits into a short.

```
short myNewShortValue = (-128 / 2);
```

And finally, this code works because we tell the compiler we know what we're doing by using this cast, and the compiler doesn't give an error.

```
short myNewShortValue = (short) (myMinShortValue / 2);
```

Primitive Types Challenge

Your challenge is to create four new variables:

- A **byte** variable, set it to any valid **byte** number, it doesn't matter.
- A **short** variable, set it to any valid **short** number.
- An **int** variable, set it to any valid **integer** number.
- Lastly, create a variable of type **long**. Make it equal to 50,000 plus 10 times the sum of the **values of the first 3 variables (your byte, your short and your int values). In other words, use the variable names in your expression to calculate the sum.**

Using Parentheses

Parentheses are another way to make your code more readable.

```
longTotal = 50000L + (10 * sumOfThree);
```

They also make it clear which calculation should be done first.

Recap

So that's it, for bytes, shorts, integers and longs, which are the first four primitive types in Java's list of available types.

We still have four left to go.

In the next video, we are going to start talking about decimal numbers, because, of course, we've only been dealing with whole numbers up until now.

Floating-point Numbers

Unlike whole numbers, floating-point numbers have fractional parts that we express with a decimal point.

In this table, you can see some examples of both whole numbers and floating point numbers, in comparison.

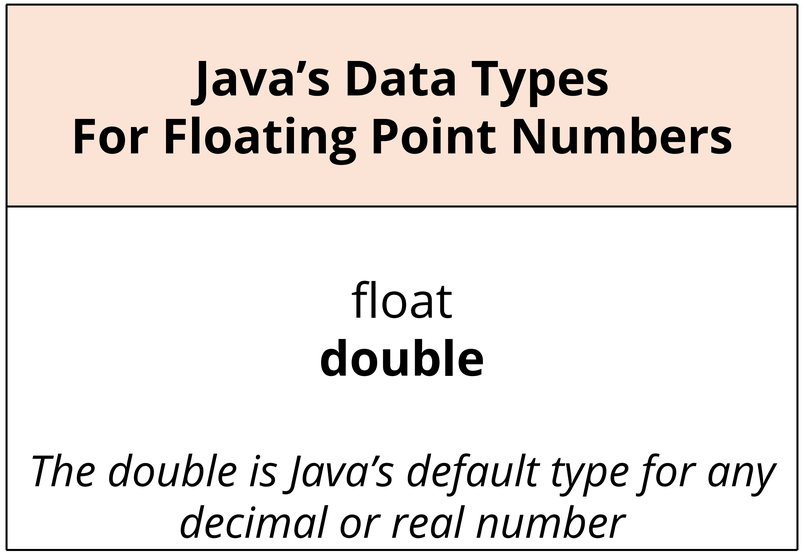
Whole number Examples	Floating Point Examples
3	3.14159
100000	10.0
-2147483649L	-0.6666666666666667

Floating-point numbers are also known as real numbers.

Floating-point Number Data Types

We use a floating-point number when we need more precision in calculations.

There are two primitive types in Java for expressing floating-point numbers, the float and the double.



The double is Java's default type for any decimal or real number.

Single and Double Precision

Precision refers to the format and amount of space occupied by the relevant type.

This table shows the width of each of the floating point types and their ranges.

The ranges are shown in Java's scientific notation, which we show below in blue color.

Data Type	Width (in bits)	Min Value	Max Value
float	32	1.4E-45	3.4028235E38
double	64	4.9E-324	1.79769313486231 57E308

You can see the e-notation followed by either a positive or negative number.

Java's Scientific Notation

Scientific notation can be translated into more familiar terms, by replacing the 'E' in the number, with the phrase 'times 10 to the power of'.

1.4E-45 is the same as 1.4×10^{-45} and 3.4E38 is the same as 3.4×10^{38}

Data Type	Min Value	Max Value
float	1.4E-45	3.4028235E38

So we can say, the minimum value of a float is $1.4 * 10^{-45}$ and its maximum value is approximately $3.4 * 10^{38}$.

Java's Scientific Notation

Think about that for a moment, using the double's minimum value shown below, remembering that 10-1 = 0.1 and 10-5 = 0.00001 for example.

Imagine writing out the double data type's minimum value in decimal format! That would be a lot of zeros after the decimal.

Data Type	Min Value	Max Value
double	4.9E-324	1.7976931348623157E308

I hope you can see that a double, when compared to a float, can represent both a much smaller decimal value, and a much larger decimal value. This is why its called more precise.

Because it's more precise, the double is the default type for floating point numbers.

float and double and numeric literal suffixes

Important: The double data type is Java's default type for real numbers.

- For example, any number with a decimal is a double.
- So, 10.5 is a double by default in Java.

- The double data type can be specified as a numeric literal with a suffix of either lowercase 'd', or uppercase 'D', but because doubles are the default in Java, the suffix is optional to use.
- On the other hand, the float data type can be specified as a numeric literal with a suffix of lowercase 'f', or uppercase 'F'. This suffix is required if you are assigning a real number to a variable that was declared with a float type.

Challenge

Thinking back to our discussion on casting, how do you think you'd do the same for the float to remove this error? I am talking about using casting here, specifically, because, as you have learned, we could just use the suffix f to tell Java this is a float. Here I want you to use casting.

```
jshell> float myOtherFloatValue = 5.25;
|  Error:
|  incompatible types: possible lossy conversion from double to float
|  float myOtherFloatValue = 5.25;
|                            ^--^
```

Certification Exam Pointer

Not everyone realizes that Java's default data type for a decimal literal is a double, which is larger and more precise than a float.

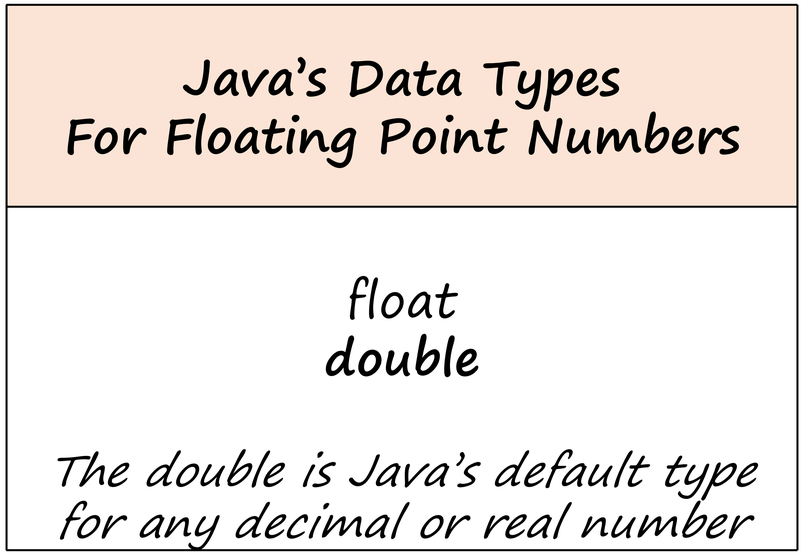
Java likes to put a similar line of code in its code segments on exam questions, to what we saw earlier, omitting that 'f' suffix. Without a computer to check, this statement can look fairly innocuous.

```
float myOtherFloatValue = 5.25;
```

The number 5.25 is a double, so assigning it to a float will raise an error.

This is a gift question to an exam taker, if you can easily spot this compiler error.

Floating Point Data Types



Default output for numeric data types

In this slide, we show default output, as it would be in JShell or by using System.out.print, for both whole and real numbers.

Whole Number Examples

Literal Value	Default Output
5	5
500_000_000_000L	500000000000

Floating Point Examples

Literal Value	Default Output
5	5.0
5.000000	5.0
5f	5.0
5d	5.0
5e1	50.0
5_000_000.0	5000000.0
50_000_000.0	5.0E7

You can see from this table, that there are more ways to express a decimal real number literal, than a whole number, with the use of the 'F' or 'D' suffix, as well as using Java's scientific notation in the literal value.

Default output for numeric data types

Another interesting difference is that, for whole numbers, the output is never in scientific notation, but for real numbers, it could be, as you can see.

Whole Number Examples

Literal Value	Default Output
5	5
500_000_000_000L	500000000000

Floating Point Examples

Literal Value	Default Output
5	5.0
5.000000	5.0
5f	5.0
5d	5.0
5e1	50.0
5_000_000.0	5000000.0
50_000_000.0	5.0E7

Why is the double a better choice in most circumstance

Why should we choose double?

First, it's actually faster to process on many modern computers.

That's because computers have, at the chip level, the functionality to actually deal with these double numbers faster than the equivalent float.

Second, the Java libraries that we'll get into later in the course, particularly math functions, are often written to process doubles and not floats, and to return the result as a double.

The creators of Java selected the double because it's more precise, and it can handle a larger range of numbers.

Challenge

The objective of this challenge, is to convert a given number of pounds to kilograms.

STEPS:

1. Create a variable with the appropriate type, to store the number of pounds that we want to convert into kilograms.
2. Calculate kilograms, using the variable above, and store the result in a 2nd appropriately typed variable.
3. Print the result.

Don't forget to use the conversion formula shown here:

1 pound is equal to 0.45359237 of a kilogram.

Floating Point Number Precision Tips

In general, float and double are great for general floating point operations.

But neither should be used when 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.

String Literal Example

If you recall, we've used literal strings before, and that's where we've typed some text in double quotes.

```
jshell> System.out.print("Hello World");
```

We haven't really used a String variable yet, but we'll be doing that in upcoming videos.

Comparing the char to the String

This table is a quick summary of the differences between the char and the String.

char	String
• Holds one, and only one, character • Literal enclosed in Single Quotes	• Can hold multiple characters • Literal enclosed in Double Quotes

Is there a good use for the char data type in today's computing world?

Why would you want to use a variable that only allows you to store one character?

One example might be to store the last key pressed by a user in a game.

Another example might be to loop programmatically through the letters in an alphabet.

char Data Type

A char occupies two bytes of memory, or 16 bits, and thus has a width of 16.

The reason it's not just a single byte, is that a char is stored as a 2 byte number, similar to the short.

This number gets mapped to a single character by Java.

- So, when you print a char, you will see the mapped character, and not the representative number.
- And you can use single quotes and a character literal to assign a value to a char, which is much simpler than looking up the representative number.

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've got 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.

Assigning values to a char variable

There are three ways to assign a value to a char: Each of these methods, represents storing the letter, capital D, in memory.

Assignment Type	Example Code
a literal character	`char myChar = 'D';`
a Unicode value	`char myChar = '\u0044';`
an integer value	`char myChar = 68;`

The char Challenge

Create three char variables to store the character for the question-mark symbol.

- `mySimpleChar` should be assigned the literal question-mark character ?.
- `myUnicodeChar` should be assigned the unicode value for the question-mark ?.
- `myDecimalChar` should be assigned the decimal value for the question-mark ?.
- Print all three variables in one statement, that starts with the label “My values are”.

Hint: Use the symbl.cc website

Note on JShell and UTF-8 unicode values

There are a lot more characters, which are not on the usual keyboard, that can be output by this method, for example the copyright symbol.

But, if you are testing this out on your own, using Windows, you should be aware that JShell may give you an unexpected result, because UTF-8 is not supported, by default, for command line operations.

I point it out here, in case you are being adventurous, and do encounter this problem.

Again, this is only a problem with JShell and Windows users.

This won't be a problem in Java or IntelliJ, or if you're using MAC or Linux.

Boolean Primitive Type

A boolean value allows for two opposite choices, true or false, yes or no, one or zero.

In Java terms, we've got a boolean primitive type, and it can be set to two values only, either true or false.

The wrapper for boolean is Boolean with a capital B.

Why would you start your boolean variable name with the prefix 'is'?

Developers will often use the word, is, as a prefix for a boolean variable name.

This creates a name that seems to ask a question, which makes reading the code more intuitive.

But other prefixes can be just as valid.

Why would you start your boolean variable name with the prefix 'is'?

Here are some example boolean variable names, such as isMarried and hasChildren, that clearly define what condition is being tested:

Boolean variable name examples
isCustomerOverTwentyOne
isEligibleForDiscount
hasValidLicense
isMarried
hasChildren

Primitive Types Recap and the String Data Type

In the previous video, we looked at the **char**, and also the **boolean** types, which were Java's seventh and eighth data types.

So at this point, you should be familiar with all eight of Java's primitives.

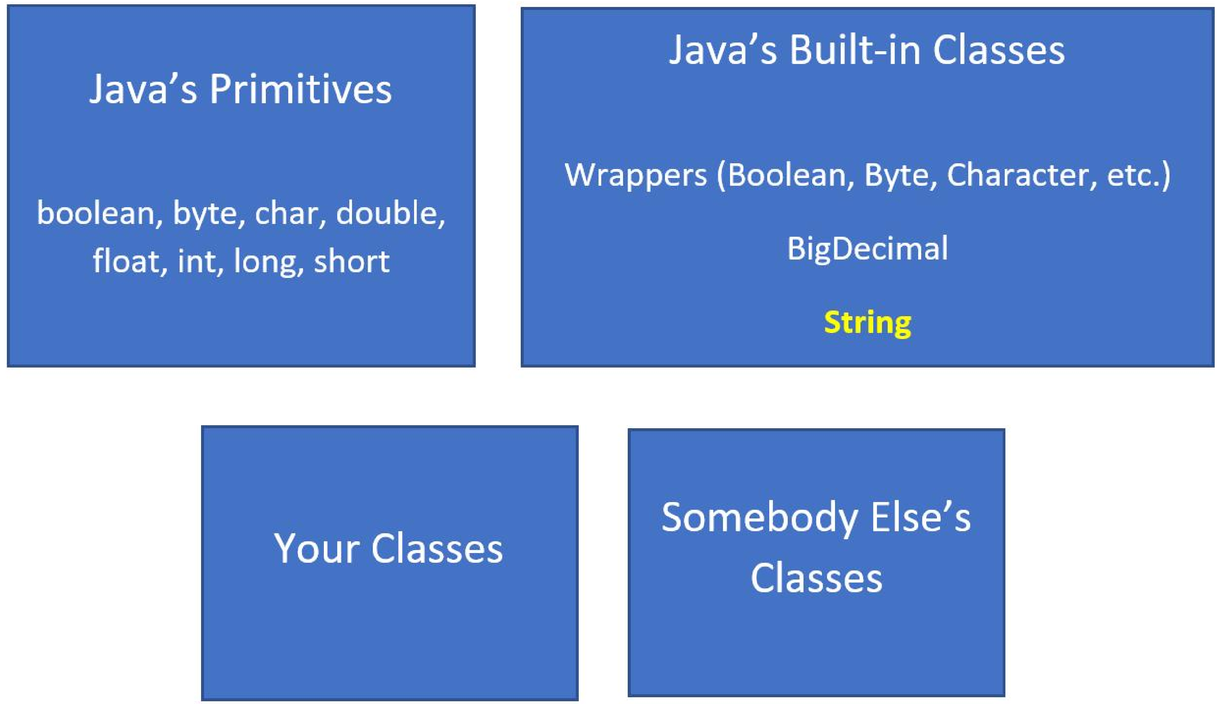
Java's 8 Primitive Data Types

Whole number	**Real Number (floating point or decimal)**
byte short **int** long	float **double**
Single character	**Boolean value**
char	boolean

The int and a double are Java's default data types for numeric literals

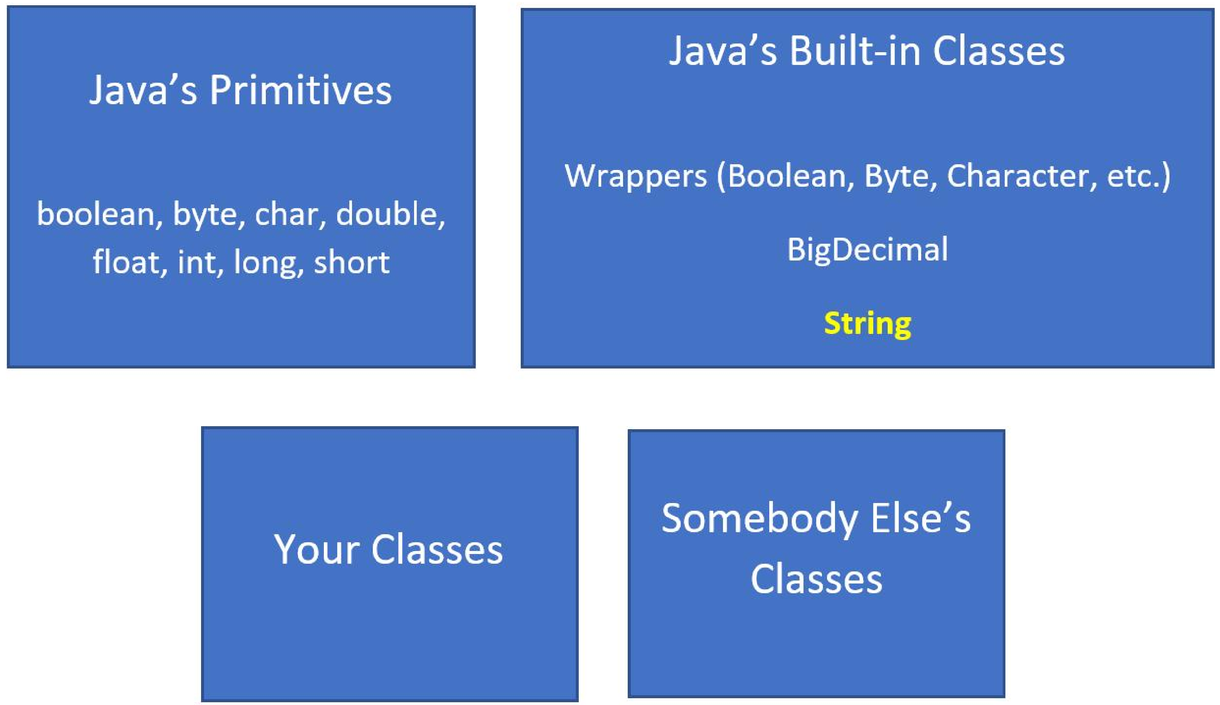
Handling Data in Java

This slide demonstrates that most Java programs use some combination of the data types shown in this diagram.



Remember, I've said that classes in Java are custom data types

Handling Data in Java



Remember, I've said that classes in Java are custom data types

You'll use Java's primitive data types, Java's built-in classes, and probably some combination of your own custom classes and classes created by other people.

So What is a String?

A String is a class that contains a sequence of characters.

Executing multiple lines of code in JShell

To execute multiple lines of code as a set, in JShell, first start with an opening curly brace and press enter.

```
jshell> {        // start with a curly opening brace
   ...>     first_statement;
   ...>     second_statement;
   ...>     third_statement;
   ...> }        // end with a curly closing brace
```

JShell will display an alternate prompt as you can see, three dots and a greater than sign.

You can add a statement and press enter, until you've added as many statements as you want to run.

Finally, add the closing curly brace, noting that a semicolon is not required after this brace.

Once you press enter after the closing brace, all of your statements will run in the order you put them.

Executing Multiple Statements In JShell

There are two ways to execute multiple statements in JShell:

- Put your statements on a single line.
- Or, enclose your statements in a set of curly braces {}.

String concatenation

In Java, the + symbol is an operator which can mean addition, if used for numbers.

But it also means concatenation when applied to a String.

A String + anything else, gives us a String as a result, concatenating anything after the String as text to the initial String.

Strings are Immutable

Immutable means that you can't change a String after it's created.

So in the case of the code we've written, the value 120.47 is technically not appended to the current contents of **lastString**.

```
lastString = lastString + doubleNumber;
```

Instead, a new String is created automatically by Java. The new String consists of the previous value of **lastString**, plus a textual representation of the double value 120.47.

The net result, is that our variable, **lastString**, has the concatenated value. However, Java created a new String in the process, and the old one will get discarded from memory automatically.

String vs StringBuilder

- The String class is immutable, but can be used much like a primitive data type.
- The StringBuilder class is mutable, meaning it can be changed, but does not share the String's special features, such as being able to assign it a String literal or use the plus operator on it.

Both are classes, but the String class is in a special category in the Java language.

The String

- The String is so intrinsic to the Java language, it can be used like a 9th primitive type.
- But it's not a primitive type at all, it's a class.

Operators, Operands and Expressions

In the previous video, we did a recap of the primitive data types, as well as getting a formal introduction to the String.

In this video, we're going to take a look at operators, operands, and expressions.

I've discussed very briefly what operators are, and we've used a few of them in expressions.

Let's go deeper into that now.

What are Operators?

So what are operators?

Operators in Java are special symbols that perform specific operations on one, two, or three operands, and then return a result.

In the example below, which we saw in a previous video, we used the addition operator, as well as the multiplication operator.

```
long longTotal = 50000L + 10L * (byteValue + shortValue + intValue);
```

But there are many other operators in Java.

What are Operands?

So what is an operand?

An operand is a term used to describe any object that is manipulated by an operator.

So if we consider this:

```
int myVar = 15 + 12;
```

The plus here is the operator, and 15 and 12 are the operands. Variables used instead of literals can also be operands.

Another example from a previous challenge::

```
long longTotal = 50000L + 10L * (byteValue + shortValue + intValue);
```

In the line above, **byteValue**, **shortValue** and **intValue** are operands, as are the numeric literals.

What are Expressions?

What's an expression?

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

They are a way of forming and combining those values to produce a result.

In the line below, 15 plus 12 is the expression, which returns the value of 27.

```
int myVar = 15 + 12;
```

What are Expressions?

In the statement below, `byteValue + shortValue + intValue` is the expression.

```
int sumOfThree = byteValue + shortValue + intValue;
```

How may operators are in this code?

How many operators would you say, are in this statement?

```
int result = 1 + 2;  // 1 + 2 = 3
```

Well, we've actually got two, the equal operator, and the plus operator.

What is a Comment?

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

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

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

What is the effect of this code on the value in previousResult?

```
int result = 1 + 2;
int previousResult = result;
result = result - 1;
```

Before we move on, some of you may be wondering about the value that is now in the **previousResult** variable after the last statement.

To summarize, we assigned result to **previousResult**, and then we changed the value of **result**. But, did this also change the value in **previousResult**? That is a good question.

The + Operator on character data types

char	String
• Holds one, and only one, character • Literal enclosed in Single Quotes	• Can hold multiple characters • Literal enclosed in Double Quotes

The + Operator on char

You might remember that we said chars are stored as 2 byte numbers in memory.

When you use the addition operator with chars, it is these numbers in memory that get added together.

The character values don't get concatenated.

The Remainder Operator

The remainder operator is represented in Java by the % sign.

The remainder operator goes by several other names: modulus, modulo or just plain mod for short.

The remainder operator returns the remaining value from a division operation.

If there is no remaining value, the result is 0.

The Remainder Operator

This table shows some examples.

Division Result	Remainder Result	Explanation
10 / 5 = 2	10 % 5 = 0	Ten can be divided evenly by 5, so there is no remainder.
10 / 2 = 5	10 % 2 = 0	Ten can be divided evenly by 2, so there is no remainder.
10 / 3 = 3	10 % 3 = 1	Ten cannot be divided evenly by 3, but we get 3 from the division which gives us 9 with 1 remaining.
10 / 1 = 10	10 % 1 = 0	Using 1 on the right side of the remainder operate will always give a result of 0.

Summary of Operators

This tables shows the five operators we just reviewed. For all of the numeric types (whole numbers and decimals), the operators are mathematical operators as shown.

Operator	Numeric types	char	boolean	String
+	Addition	Addition	n/a	Concatenation
-	Subtraction	Subtraction	n/a	n/a
*	Multiplication	Multiplication	n/a	n/a
/	Division	Division	n/a	n/a
%	Remainder (Modulus)	Remainder (Modulus)	n/a	n/a

Summary of Operators

Only one operator, the + operator, is supported by String, but it means concatenation, not addition.

Operator	Numeric types	char	boolean	String
+	Addition	Addition	n/a	Concatenation
-	Subtraction	Subtraction	n/a	n/a
*	Multiplication	Multiplication	n/a	n/a
/	Division	Division	n/a	n/a
%	Remainder (Modulus)	Remainder (Modulus)	n/a	n/a

Summary of Operators

None of the operators are applicable to a boolean.

Operator	Numeric types	char	boolean	String
+	Addition	Addition	n/a	Concatenation
-	Subtraction	Subtraction	n/a	n/a
*	Multiplication	Multiplication	n/a	n/a
/	Division	Division	n/a	n/a
%	Remainder (Modulus)	Remainder (Modulus)	n/a	n/a

Summary of Operators

Because the char is stored as a whole number literal, all the operations are applicable to a char.

Operator	Numeric types	char	boolean	String
+	Addition	Addition	n/a	Concatenation
-	Subtraction	Subtraction	n/a	n/a
*	Multiplication	Multiplication	n/a	n/a
/	Division	Division	n/a	n/a
%	Remainder (Modulus)	Remainder (Modulus)	n/a	n/a

Abbreviating Operators

In the previous video, I talked about operators, operands, and expressions. In this video, I'll continue the discussion about operators, and you'll see how they can be abbreviated to simplify code.

In the next section of this course, we'll be transitioning to coding in a code editor called IntelliJ. I want to continue to prepare you for that environment by using the curly brace feature in JShell.

Why do we want to use multiple statements in curly braces {}?

So, why use multiple statements in curly braces?

- First, it's a way to group statements together before executing them.
- It allows us to put statements on multiple lines which is more natural and readable.
- We can execute the group of statements as a whole, which more closely resembles running code in Java.

Incrementing by One

Incrementing by one is a very common requirement in programming.

Obviously we can do the following:

```
result = result + 1;
```

But we also have two other shorthand ways to do this same thing.

Shorthand (or Abbreviating) Operator	Code Sample
Post-fix Increment Operator	result++;
Compound Assignment Operator with + sign	result+=1;

Decrementing by One

Decrementing by one is also very common.

We can decrement simply by using the equation:

```
result = result - 1;
```

But we also have two other shorthand ways to do this same thing.

Shorthand (or Abbreviating) Operator	Code Sample
Post-fix Decrement Operator	result--;
Compound Assignment Operator with - sign	result-=1;

Compound Assignment Operator Challenge

Using the code we have been using, either by scrolling up and editing the group of statements, or creating a new group.

- Initialize an **int** variable, named **result**, to the value of 10, rather than 1.
- Next, use the compound assignment operator, with the minus sign, to subtract a number from **result**, using a value of your choice.
- Print the result out, using the System.out.print statement.

Compound Operator Assignment

When **result** is an **int**, the compound operator assignment

```
result -= 5.5;
```

give us a different result from what we expected, which was

```
result = result - 5.5;
```

Compound Assignment Operator

The compound assignment operator

```
x -= y
```

is often said to be

```
x = x - y
```

but that's not entirely true if y is not the same data type as x.

Compound Assignment Operator

```
x -= y
```

is really

```
x = (data type of x) (x - y)
```

An implicit cast is done when using this operator, so no error occurs, but unexpected results may happen, as we have seen.

Compound Assignment Operator

So, summarizing, for our own sample of code:

```
result -= 5.5;
```

is really

```
result = (int) (result - 5.5);
```

The abbreviating operators

The abbreviating operators we've discussed so far are:

Shorthand Operator	Code Sample
Post-fix Increment Operator	result++;
Post-fix Decrement Operator	result--;
Addition Compound Assignment	result += 5;
Subtraction Compound Assignment	result -= 5;
Multiplication Compound Assignment	result *= 5;
Division Compound Assignment	result /= 5;