3. Variables, Data Types, and Math Operators

Contents

[1. Introduction 1](#_Toc24167)

[2. Variables 2](#_Toc25990)

[3. Primitive Data Types 2](#_Toc26207)

[4. Primitive Types Are Stored by Value 2](#_Toc8936)

[5. Arithmetic Operators 2](#_Toc27398)

[6. Prefix, Postfix, and Compound Assignment Operators 2](#_Toc10413)

[7. Operator Precedence 2](#_Toc16420)

[8. A Closer Look at Operator Precedence 2](#_Toc4399)

[9. Type Conversion 2](#_Toc32172)

[10. A Closer Look at Type Conversion 2](#_Toc26507)

[11. Summary 2](#_Toc29019)

# 1. Introduction

=>slides: Pg. 1-2

Welcome to our next module, Variables, Data Types, and Math Operators. In this module, we're going to look at the capabilities that Java provides to allow us to manage and represent data within our applications. So now to do that, the first thing we need to understand are variables. How do we represent data values within our application? From there, we're going to look at what we call primitive data types. Primitive data types are the foundation of Java. These are the fundamental data types that underlie everything we do within our Java application. As part of that discussion, we're going to look at how the Java compiler represents primitive data and how it manages the storage when you assign values from one variable to another. Now, of course, as we work in our applications, we're going to need to do arithmetic. So we're going to take a look at the arithmetic operators within Java. And as part of this discussion, we'll look at operator precedent. How does Java decide what order to perform the operations when you're dealing with a complex equation? And then, with so many data types inside of Java, we need to look at data type conversions. We have a situation where we have an operation or equation that involves multiple data types. In that situation, we want to understand what conversions the Java compiler will perform for us automatically, as well as how we can take control that process and explicitly make conversions.

# Variables

=>slides: Pg. 3

In order to do anything interesting in our programs, we need a way to store and manipulate values. That's where variables come in. Now a variable is simply a named area of storage. Now Java is a strongly typed language, so when we declare a variable, we need to indicate its type. For example, this variable, dataValue, has a type of int, which means it can only be assigned values that are compatible to type int, so if it's a value of 100, which is an integer. Now when we work with variables, we can declare and assign the variable in separate statements, as we do with dataValue, or we can do it all in one step. So our variable, myInfo, is of type int, and as part of its declaration, we've set its initial value to 200.

=>slides: Pg. 4

By convention, variable names consist of only letters and numbers. Technically, the Java compiler allows the use of some special symbols, but in practice, we use just letters and numbers. For example, our variable here, total, is valid because it consists of only letters, and the variable, grade4, is valid because it's made up of only letters and numbers. Now one thing to keep in mind. Variable names cannot start with a number. So this variable, 2much, is invalid. The compiler will actually raise an error for any variable name that starts with a number.

=>slides: Pg. 5

When we name our variables, we generally use a naming style called Camel Case. And Camel Case simply means that when declaring a multi‑word variable name, we start each word after the first word with an uppercase letter. All other letters are in lowercase. So the variable, sum, is all lowercase because it consists of only a single word. The variable name, studentCount, is made of two words, so the second word, Count, starts with an uppercase C. All other letters are lowercase. Another example, bankAccountBalance. The second and third words, Account and Balance, each start with an uppercase letter. And one last example, level2Training. The second word, Training, starts with an uppercase letter. So all four of these examples correctly conform to the Camel Case naming style.

=>slides: Pg. 6

Let's take a quick look at some examples of using variables. So we'll start by declaring a variable, myVar. Now, at this point, myVar is considered an uninitialized variable, which means it's been declared but does not yet have a value, so we can't yet use it. But once we assign a value to myVar, we're then free to use it. So let's go ahead and print out myVar, which, of course, displays the value 50. Let's go ahead and declare a variable, anotherVar. Now as part of the variable declaration, we give anotherVar an initial value 100. And since we gave anotherVar a value as part of the declaration, we can use it right away. So printing anotherVar displays the value 100. Java allows us to modify the value of variables. So we can do things like assign the value of anotherVar to myVar. So now when we print out myVar, rather than displaying the value 50 as it did earlier, it now displays the value 100.

=>slides: Pg. 7

In some cases, we may not want to allow changes to a variable, and that's where the final modifier comes in. So we've declared this variable, maxStudents, to have an initial value of 25. We've marked the variable as final, which means that once the variable's value is set, it cannot be changed. When marking a variable as final, we do not have to assign the value to the variable right away, which is the case with this variable, someVariable. And this leaves us free to do whatever other work in our program is necessary, including declaring other variables. And then when we're ready to assign a value to someVariable, we can do that. Now once the value of someVariable is set, the compiler makes sure that we don't later try to modify it. And this is a really powerful capability because it helps avoid areas that occur due to a variable being accidentally modified, which in complex applications can occur more easily than it might seem. Okay, so next, let's take a look at some of the data types we can use when declaring variables.

# Primitive Data Types

=>slides: Pg. 8

As we've seen, when we declare our variables, we associate those variables with a type, and some of the most common data types we use are what are called the primitive data types. Now the word primitive may give you the indication that these types are somehow less than, they're somehow not the really good data types. But that's not true. It's simply the name of the data types that are built into the Java language. And these primitive data types are the foundation of all other types. Now there are four categories of primitive data types. So there's the integer types, the floating point types, the character type, and the Boolean type.

=>slides: Pg. 9

So let's look first at the integer types, and there are several different integer types, but the only difference between each one is the range of data that it can store and the size that it takes up. So the smallest center type is what's called the byte type, and the byte type takes up simply 8 bits. And because it's so small, it can only store values between ‑128 and positive 127. So if I declare a variable here, numberOfEnglishLetters, its type is byte, and I'm giving it an initial value of 26. And we also have a short integer type. That takes up 16 bits, and because it's larger it can store a larger range of values, so from ‑32,000 to positive 32,000. So we declare a variable here feetInAMile, its type is short, and i give it a value of 5280. Then we have the int type. Now int is probably the most commonly used integer data type. Its size is 32 bits, and it can store between ‑2 billion and positive. 2 billion. So I have a variable here, milesToSun, its type is int, and I assign it an initial value of 92,960,000. And then we have our largest integer type, which is a long, and that's a 64‑bit integer. Now as you can see, that can store really, really large positive values or really, really small negative values. Now the thing to notice about longs is the literal format. When you want to set a long literal, you have to put the letter l at the end of the value. So I have a variable here, milesInALightYear, its type is long, and when I give it that initial value to indicate that that literal is a long, I add the letter l at the end of it. And that tells the compiler that this is a literal of type long.

=>slides: Pg. 10

Next, we have our floating point types, and floating point types simply have the ability to store values to have a fractional portion. In other words, they store decimal values. And there are two floating point types. The smallest one is what's called a float. That's a 32‑bit value, and you can see the value range there. The key thing to note about floats is that when we specify literals, they have to have the letter f after them. So if I declare a variable here kilometersInAMarathon, its type is float, I set it to 42.195, then we use the letter f to indicate that this literal is of type float. Now floating point types can have both positive and negative values, so I have a float here, absoluteZeroInCelsius, I give it a value of ‑273.15. Again, I used the F to indicate that it's a float literal. Now in addition to the float type, we also have double, and the double, again, can contain a fractional portion, but you notice it takes up 64 bits. So it allows you to have a much larger range of values. When you specify the literals for a double, you can simply specify a literal that contains a decimal. In that case, it's automatically a double. Or you can explicitly indicate that it's a double by putting a d at the end of it. So you can see I have a variable here, atomWidthInMeters. Its type is double. I give it an initial value of that 0 point, a whole bunch of 0s and number 1. Then I use the letter d to explicitly indicate that it's a double.

=>slides: Pg. 11

Now Java also has what's called the character data type, and the character data type stores a single Unicode character. Note that it's a single character; it's not a string of characters. We'll talk about strings a little bit later in the course. Now when we specify the character types, the literals are placed within single quotes. So you notice here I declare a variable whose name is regularU, its type is char because char is the name of the type that's used to store a single character. And we give it that initial value, the letter U enclosed between single quotes. Now note that this type stores Unicode characters, so it stores the full range of Unicode characters, which means it can store values that aren't necessarily on your keyboard. So because of that, we can actually use what are called Unicode code points to specify a value. And that simply means we can use the numeric representation of a Unicode character. We do you that by using a \u, followed by the four‑digit hex code. So here I have a variable name accentedU. It's type is character, and I want it to store the capital U with an accent. I don't have that on my keyboard, so I can simply specify its value is \u00DA, the hexadecimal representation of that character, and all that's enclosed in single quotes. So when I display this with in my application, I actually get that uppercase U with an accent.

=>slides: Pg. 12

And then finally we have the Boolean type. The Boolean type is simply a type that can store true and false values. Its literals are the word true and the word false. So here I've declared a variable I love Java, its type is Boolean, and I give it an initial value of true. Okay, so that's our primitive data types. So in our next clip, let's take a look at how Java manages the storage of our primitive data types.

# Primitive Types Are Stored by Value

=>slides: Pg. 13

When we work with primitive types in Java, those types are stored by value. That means that each variable gets its own independent copy of each value. So we have a variable here called firstValue. We give it an initial value of 100. That means the Java allocates an area of memory called firstValue and that value of 100 is placed into that area of memory. So now, if we declare another variable otherValue and initialize it to be firstValue, that means another area of storage called otherValue is created, and the value that's inside of firstValue is copied over to otherValue, so otherValue also has a value of 100. But even though these two variables have the same value, the value's themselves are independent. There's no relationship between those two variables after the assignment is made. So if we now go to firstValue and set it to a value of 50, it means firstValue loses the value of 100 and replaces it with a value of 50, but otherValue is completely unaffected. So in otherValue, if we go ahead and assign a value of 70, otherValue replaces the value of 100 with that value of 70, and firstValue is completely unaffected. So the key thing to understand here is that when we work with primitive types in Java, each variable's values are unrelated to any other variable's values. When we make an assignment, there is an independent copy made of that value. The reason that's important to understand is because if we go through and learn more about Java throughout this course series, we'll see that there are other data types where that storage is managed differently. Okay, so now in our next clip, let's take a look at some of the arithmetic operators within Java.

# Arithmetic Operators

=>slides: Pg. 14

As we build our applications, we're commonly going to have to do calculations within those applications. In order to our calculations, we'll need to use arithmetic operators, and Java has a number of categories of arithmetic operators. The first category, it's what we call the basic operators, and these are generally what you think of as arithmetic operations, things like adding, multiplying, dividing, that sort of thing. And these arithmetic operators work very much like they do in traditional mathematics. You can simply apply the operators, and they'll produce a result. When you apply the operators, the values that are operated on are not affected by the operation. You simply get a result from the operation. But Java also has operators to behave a bit differently. For example, we have the prefix and postfix operators, and these are operators that simply increment or decrement a value by 1. The key thing is when you apply these operators to a variable, they actually replace the variable's original value, and these are not the only operators that replace the value. There's also what are called the compound assignment operators. These also operate on a value, perform some calculation with that value, but at the end of the calculation, they replace the variable's original value with the result of the calculation.

=>slides: Pg. 15

So let's first take a look here at the basic operators. And these, again are the operations you would generally expect, things like add, subtract, multiply, and divide. And in some cases, operators will behave differently when it comes to floating point values versus integer values. Now initially, they behave the same. So, for example, in the case of adding, if you add 1 and 2 together, you get a result of 3, whether you're dealing with floating points or integers, and the same thing with subtraction. If I subtract 4 from 5, I'll get 1, whether I'm doing floating points or integers, and the same sort of thing with multiplication. Multiplying 4 times 2 gives me a result of 8. But then when you get to division, things start to behave a bit differently, depending on the data type. So if I'm dealing with floating point, if I divide a value 13 by 5, I'll get a result of 2.6, 5 goes into 13 2.6 times. But when it comes to doing division with integers, remember that integers can't have a fractional portion. So when you do division with integers, the result is always the whole number of times the division can occur, 5 goes into 13 2 whole times. It simply discards the fractional portion, so there's no concept of rounding or anything like that with integer division. It's simply the whole number of times the division can occur. Now the last operator is one you might not be familiar with. It's called the modulus operator. What the modulus operator does is it gives us the remainder as a result of the division. So if I take 13 modulus 5, well, 5 goes into 13 2 whole times, 2 times 5 is 10, so the remainder is 3. So that's the result of performing a modulus operation. Now floating points can also do modulus operations. When you do a modules operation with floating point, it behaves very similar to the way it does with integers. If I say 13.0 modulus 5.0, I'll again get 3, much like what I got when working with integers. So now in our next clip, let's take a look at the other arithmetic operators.

# Prefix, Postfix, and Compound Assignment Operators

=>slides: Pg. 16

Then we have our prefix and postfix operators. So this is the increment operator, which is ++. It increments a value by 1. When you increment a variable, it takes its original value, increases it by 1, and takes the result and puts it back into the variable. Similarly, we have the decrement operator, which is ‑‑. It reduces the value by 1. Now the order you apply one of these operators affects its behavior. If you place the operator before a variable, that's considered using it as a prefix operator, and in that case, it will apply its operation and then return back the result of performing the operation. But if you take the same operator and you place it after a variable, that's considered a postfix operator. In that case, it first returns back the variable's original value and then applies the operation. So now the best way to understand the difference is to take a look at some code.

=>slides: Pg. 17

So let's declare a variable here called someValue and we'll initialize it to 5. Then I'm going to print out increment someValue. So in this case, the increment operator is used as a prefix operator. It's a prefix operator because it appears before the variable. Well, in this case, we'll take the value 5, increment it by 1, which makes it 6, and then return back that result. So the result of this print statement would be the value 6. But keep in mind that that variable, someValue, is actually changed. So if I now simply print out someValue, I will again get 6. So the operator applied the operation and stored it back into that same variable. But now let's take a look at another example. Let's declare another variable, someOtherValue. We'll again initialize it as 5. But this time when we print it out, we're going to use the increment operator as a postfix operator. In other words, we're using it after the variable. Well, in this case, we first return back the variable starting value and then apply the operation. So the result of this print statement is simply going to be 5, but someOtherValue is still incremented. So if I print it out now, I'll get back that result of 6. So the key issue there is that when the operator's before the variable, you'll get the result of the operation. When the operator's after the variable, you'll get back the value prior to the operation, but the operation is still performed.

=>slides: Pg. 18

Then, finally, we have the compound assignment operators. And as their name implies, they combine an operation and an assignment. So what these do is they apply the right side of a calculation to the left side of the operator, and then they take that result and store it back into the variable that's on the left side of the operator, and these operators are available for the five basic math operations. So there's a compound operator for addition, subtraction, multiplication, division, and modulus. Now in my experience, I use the compound assignment operators for addition and subtraction far and away most often. But there are some occasions where the other compound assignment operators are also useful. So let's take a look at some code that uses these operators.

=>slides: Pg. 19

So I'll declare a variable, myValue, giving it an initial value of 50. Then I'm going to say myValue ‑= 5. So what this operator does is it takes the value that's in myValue, which is 50, subtracts 5 from myValue, and then stores that result back into myValue. So when we print out myValue, we get 45. So the operator used myValue as part of the equation, and then when it produced a result, took that result and put it back into myValue. Now we can use these compound assignment operators of equations of any complexity. So let's take a look at another example.

=>slides: Pg. 20

So I'll declare another variable, myOtherValue, that I initialize to 100. Then I'll declare two more variables, val1 and val2. Then I'll add the code, myOtherValue /= val1 \* val2. So now in order to perform this calculation, the first thing the compiler has to do is solve the right side of the operator. So in this case, it'll multiply val1 \* val2. So that's 5 \* 10, which produces a result of 50. It then takes that 50 and divides it into myOtherValue, which currently has a value of 100. Dividing 50 into 100 produces a result of two. That result of two is then stored back into myOtherValue. So when we print out myOtherValue, we get back a result of 2. So as you can see, Java supports a really rich set of arithmetic operators. So in our next clip, let's take a look at the precedence of these operators.

# Operator Precedence

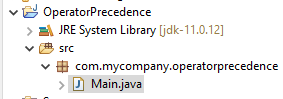
=>slides: Pg. 21

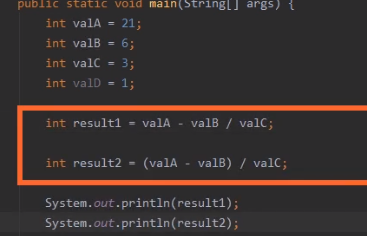
As we build our applications, we will calmly deal with calculations that involve multiple operators. So because of that, it's important we understand what order the compiler is going to work through those operators. This is a concept known as operator precedence. So the operators with the highest precedents are the postfix operators. Remember, that's the increment and decrement operators used after a value. Then we have the prefix operators, which again are increment and decrement, but in this case appearing before their values. Following that, we have the multiplicative operators, which are multiply, divide, and modulus. Then finally, we have the additive operators, which is addition and subtraction. So what this means is the postfix operators have the highest precedence, so they're done first, followed by the prefix operators, all the way down to the additive operators which have the lowest precedents.

=>slides: Pg. 22

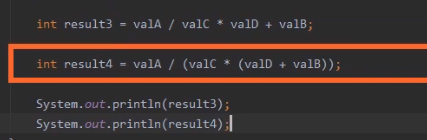
Now our calculations will often have multiple operators that have the same precedence. So in that case, the compiler will resolve them working left or right, so it still respects each precedent level, but when there's a tie within a precedence, then the work is done left to right in our calculation. Now it's important to remember that you're going to override precedents by using parenthesis, so if you enclose a portion of your calculation within parenthesis, the content of the parenthesis is done before the rest of the calculation. And you could even nest parenthesis, so you can have nesting layers of overriding the precedence. And I would really encourage you to take advantage of parenthesis. Pretty much any time I'm dealing with a complex calculation, I use parenthesis to make it very clear what order I expect the portions of that calculation to occur. So now, to get a better understanding of all this, in our next clip, let's get back into STS, and let's see how operator precedence impacts the results of our calculations.

# A Closer Look at Operator Precedence





Here we are in STS. And what we want to do now is take a look at how the order of operations impacts the results of our calculations. So you see I've got a starting application here, and we have four variables in it. And each of those variables are already initialized. So let's set up two equations. Now in both equations, we have valA ‑ valB / valC, but with one key difference. In the second equation, we've enclosed valA ‑ valB in parentheses. And remember that by using parentheses, we're able to override to default order of operator precedence. So let's run the app and see what we get. And when we run it, we can see we get two very different results. Result1 has a value of 19. Result2 has a value of 5. Let's take a look in each equation, and let's see how we got to those results. So let's start here where we assign result1. Well, the result1, we have our two operators. We have the subtraction operator, which is part of the additive operator group. Then we have our division operator, which is part of the multiplicative operator group. Well remember that the multiplicative operator group has a higher precedent. So that means that the division will be done before the subtraction. So that means it will take our valB, which has a value of 6, and valC, which has a value of 3, will do that division, so 6 / 3 is 2. Then it will subtract that from valA, which is a value of 21. So 21 ‑ 2 gave us that result of 19. So now let's look here at result2. In result2, we use parentheses. Those parentheses allowed us to override the default operator precedent. So that means in this case, the subtraction is done first. So the first thing we do is take valA, which is 21, and subtract from it valB, which is 6. 21 ‑ 6 is 15. Then we take that result and divide it by valC, which is 3. And 15 / 3 gave us that result of 5. So let's take a look at another example.

  
So to do that, we'll first scroll down a bit. We'll then set up two more equations. In both of these equations, we have valA / valC \* valD + valB. Again, we use parentheses in the second equation. In fact, we have two sets of parentheses, an inner set around valD + valB and then another set around where valC is multiplied by the result of adding valD and valB together. So let's run the app again. And you can see here this time, result3 has a value of 13. Result4 has a value of 1. So let's take a look at how we got those results. So let's take a look up here where we set the value of result3. So in this case, we have those 3 operators. All right, the first operator is division, the second operator is multiplication, and the third operator is addition. Well remember that both division and multiplication are both in the multiplicative operator group, which means they're both on the same precedence. So in that case, we do the work from left to right. So that means the first thing we're going to do is take valA, which has a value of 21. Divide that by valC, which has a value of 3. So 21 / 3 gives us a result of 7. We then take that result and multiply it \* valD. D has a value of 1, so 7 \* 1 is 7. And then we add in valB, which has a value of 6, and 7 + 6 gives that result of 13. So now let's take a look at where we assign result4. And in this case, we use parentheses and not just one set of parentheses, but two sets of parentheses with one nested inside of the other. Now remember that when we do one group of parentheses inside of another, we do the innermost one first and then work our way out. So our innermost group of parentheses indicates that we should first add valD and valB together. Well, valD has a value of 1. ValB has a value of 6. So we add 1 to 6, and that gives us a value of 7. We then move from that inner group of parentheses to the outer group. Well that means we take the value of valC, which is 3. Multiply that times the result of adding valD and valB together. So we say 3 \* 7. That gives us 21. Then we do the division. Well valA has a value of 21. So 21 / 21 gives us that result of 1. So as you can see, operator precedence directly affects the results of our calculations. And we can take advantage of parentheses to control the order that the calculations occur. Okay, so next , let's take a look at data type conversions.

# Type Conversion

=>slides: Pg. 23

Now, as we've seen, Java has a number of different data types, and there are times where we'll need to convert from one type to another. Now broadly speaking, conversions fall into two broad categories. One is what we call implicit conversions, and that's a conversion that can be automatically performed by the compiler. So when we look at an example of that, we have a variable here that's an integer. I then take that integer value and try to assign it to a long. Well, in this case, the compiler will have to convert that integer into a long to make the assignment. This is a case where the compiler's making that conversion implicitly. Now we also have what we call explicit conversions, and explicit conversions are where we tell the compiler specifically what conversion we want to have occur, and we do that using what's called the cast operator. So if I have a long variable as I have here and then I assign that over to an int, you'll notice that next to the long variable, I have the word int in parentheses. That's the cast operator. So I'm explicitly converting this long to an int using that cast operator. Now let's first look at implicit type conversions.

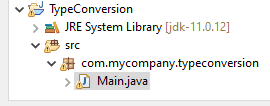
=>slides: Pg. 24

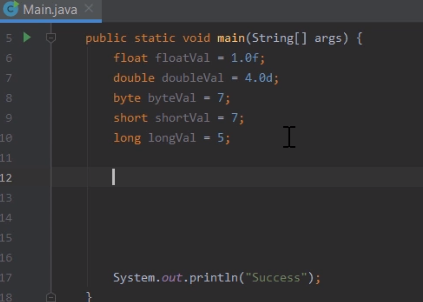
Now there are only certain kinds of conversions the compiler can perform implicitly, and these are what we call widening conversions. And conceptually what we're doing is going from one type to a wider type. And the rules for these conversions are very well defined. First, we have the issue of mixed integer sizes. Well, if we have an equation that has integers of different sizes, the composite needs to make a conversion to work with all of them as the same size. So in this situation, the compiler will use whatever the largest integer in the equation is. So if we had a short which is 16 bits and an int which is 32 bits, the compiler will implicitly convert the short to an int to use that in the equation. Now we also have the issue of mixed floating point sizes. Well, remember, there are only two floating point sizes, float and double. So any time we have mixed floating point sizes, in that equation the compiler's automatically going to use a double for all the float values. And then finally we have a mix of integer and floating point. So if we have an equation that has both integers and floating points in it, the compiler's going to use the largest floating point in the equation.

=>slides: Pg. 25

So now explicit type conversions. Again, this is a scenario where we're telling the compiler these exact conversion that we want to have occur. So that gives us control. So we can perform both widening and narrowing conversions. The thing we need to be aware of, because the compiler will do whatever we ask it to do in this situation, we need to be aware that there are some potential side effects that may occur that may not be what you want to have happen. One of the biggest one to consider is narrowing conversions, in other words, going from a wider type like a long to a narrower type like a short. Well, in that scenario, you may have a data value in that wider type, in that long, that won't fit into that short. Well, in that situation, the compiler will simply throw away any bits that don't fit. So you may end up with some unexpected behaviors in that situation. Because the data value may actually get changed as a result of the cast, because the original data value won't fit into that narrower data type. Now the next one we want to be aware of is casting a floating point to on integer. Remember that floating point values can have a fractional portion; integers can't. So when you make this conversion, any fractional portion is simply discarded. And then finally, we have casting an integer to a floating point. In this situation, you may lose precision. In other words, the value may become less accurate. And this is tied to a nuance in the way floating point values are stored. So in this situation, an integer value of say, 100,000, when it gets converted to the floating point, rather than being an even 100,000, may be something like 99,999.9999. And that's just a nuance of how floating point values are stored. Now the details of how floating points are stored is outside the scope of this course, but if that's something you're interested in, there's a readme file on the exercises folder for this module, and in there I have some URLs which will give you more detail on the behavior and storage of floating points. But for our purposes, we're staying focused on type conversions. So in our next clip, let's take a closer look at the way type conversions behave.

# A Closer Look at Type Conversion





Here we are back in STS, and what I want to do now is take a closer look at the way Java handles type conversions. So to get us started here, I've got a program in place with a few variables declared. I've got a variable of type float named floatVal, one of type double named doubleVal, then I have byteVal, shortVal, and longVal. So that gives us a good mix of integer types and floating point types that we can work with. Now, if the program runs successfully, it prints out a message of Success. But as we'll see, we won't need to run the program very often to see if we made a mistake because the STS environment gives us really good feedback to show us when we make mistakes in our code.



So the first thing we'll do here is declare a variable named result1 of type short. So we'll assign to that byteVal. Now once I do that, let's go ahead and run the application. And you can see our program printed out the message Success. So that shows us that assigning a byte to a short works just fine. And that works because a short is larger than a byte. In other words, it's a widening conversion, and we know the compiler can automatically handle widening conversions for us.



But now, if we take that byteVal that we're assigning to result1 and change that to be longVal, notice the line is underlined in red. I didn't need to build or run my program to see that there's a problem. So let me go ahead and hover over that red line. And you notice that it says that it found a long, but it requires a short. So the issue there is that a long is bigger than a short, so that would make this a narrowing conversion. And as we know, the compiler cannot automatically do narrowing conversions.



Now, that doesn't mean we can't make this assignment. It just means that we have to be explicit about it. So what we can do is explicitly cast this longVal to be a short. We do that by putting the type short within parentheses just before a longVal. And now our code is fine. By explicitly converting the long to a short, the compiler can now do it because we've indicated that's the conversion that we want to have happen.



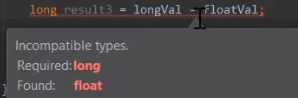
So let's try another scenario. Let's declare a result2 of type short, and we'll assign result2 the result of our byteVal ‑ longVal. Now, notice again that we're underlined in red. If I hover over that, we can see that, again, we have a long type trying to be assigned to a short. Because remember, when we mix two integer types, the largest integer type is what gets used, so since a long is larger than a byte, the result of the calculation is of type long as well, and we already know we can't assign a long into a short and have the compiler do the conversion for us. But we can take this result and cast it into a short.



So I'll put short in parentheses just before a byteVal. Now you'll notice when I add that, the line is still underlined in red because as it's currently written, the only thing I'm converting is byteVal. I don't want to just convert byteVal, I want to convert the results of the calculation.



So to do that, I'll enclose the entire calculation in parentheses. And now that's valid. The calculation is performed as a long, but then we take that long result, convert it to a short, and then can assign it to our variable of type short. So now let's go ahead and declare a variable result3 of type long.



And we'll assign this longVal ‑ floatVal. You can see this is underlined in red, meaning there's an error, and if I hover over it, notice that the required is a long, but it found the float. And of course, the issue there is that when you have a calculation that has both an integer type and a floating point type, the result is going to be that floating point type. So now we could again explicitly convert the result of this calculation to be a long, but there's some danger in doing that. Because remember, that a long as an integer type can't handle decimal places, and a floating point value might have decimal places.



So in this case, a better solution might be just to take result3 and make it a float rather than a long. So now our calculation is performed as a float and we can store that into our result3, which is also a float. So as you can see, it's really important that we understand our type conversions and when they can be done implicitly, as well as when we can do them explicitly using a cast.

# Summary

=>slides: Pg. 26

To wrap up, here are some of the key things you want to remember from this module. Remember that in our applications, we're going to need to store data values. We do that using variables. Remember that Java is a strongly‑typed language, so our variables are strongly typed. In other words, when we declare variable, we have to specify the type of data we want to store in that variable. Remember that Java normally allows variables to be modified. So you can give it an initial value and assign new values to it anytime we need to throughout the course of our program. But we do have the option of declaring a variable as final. And in that case, compiler will make sure that once we assign an initial value to that variable, it'll make sure that we do not attempt to modify the variable. Java has a rich type system. And the foundational types are what are known as the primitive types. And there are four categories of primitive types. We have the integer types, which store whole numbers. We have the floating point types, which can store values with a fractional portion. We have the character type, which can store a single character, and then the boolean type, which can store true and false values.

=>slides: Pg. 27

Now in our Java applications, when we want to do arithmetic, we're going to use math operators to do that. Now there are three categories of math operators. Now we have the basic operators, which are add, subtract, multiply, divide, and modulus. Then we have the postfix and prefix operators, which are plus plus and minus minus. Those allow us to increment or decrement a value. And then we have the compound assignment operators, and these allow us to use a value in a variable for calculation, take the result of that calculation, and put it back into that same variable. As we work with our math operators when we're performing complex equations, we need to know what order the operators are going to be executed. That's a concept known as operator precedence. Remember that Java has a well‑defined order of precedence, the highest operator precedence with the postfix operators, then prefix operators, then the multiplicative operators, and then finally, the additive operators. Now in cases where there's a tie, remember that the compiler will perform the operations from left to right. But also remember you always have the option of using parentheses to be explicit about what order you want the operations performed.

=>slides: Pg. 28

And then finally, we have the issue of type conversions. As we deal with the different data types within our application, there will be times when a conversion needs to occur. There are two broad categories of conversions. We have the implicit type conversions, which are the ones that compiler can perform for us automatically. Those are the widening conversions, going from a narrower type to a wider type. Remember, we also have explicit type conversions, and these are conversions where we tell the compiler what we want to have happen. We do that by using the cast operator. The cast operator is simply the target type enclosed in parentheses. And because we're explicit about what we want to have happen, we can do widening or narrowing conversions. But we want to be aware because the compiler will do whatever we ask it to do. There may be potential side effects. So when we do explicit cast, we want to be sure we know what those side effects are. Alright, that wraps up this module. In our next module, we'll take a look at conditional logic and block statements.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*