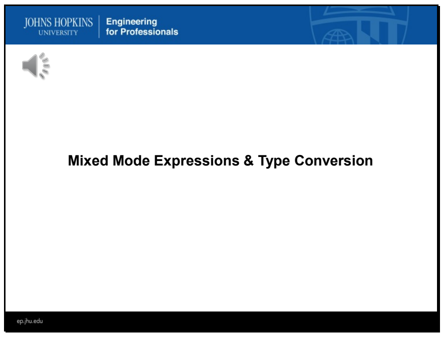
1



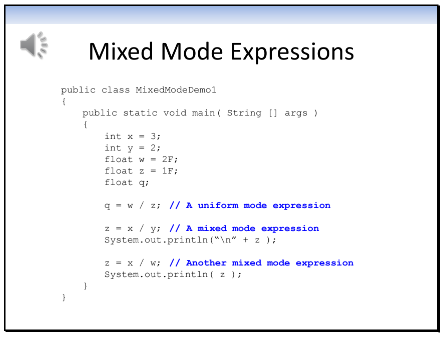
JOHNS HOPKINS UNIVERSITY | Engineering for Professionals

Mixed Mode Expressions & Type Conversion

ep.jhu.edu

In this lecture you will learn about how Java handles mixed mode expressions and type conversion.

2



Mixed Mode Expressions

```
public class MixedModeDemo1
{
    public static void main( String [] args )
    {
        int x = 3;
        int y = 2;
        float w = 2F;
        float z = 1F;
        float q;

        q = w / z; // A uniform mode expression

        z = x / y; // A mixed mode expression
        System.out.println("\n" + z );

        z = x / w; // Another mixed mode expression
        System.out.println( z );
    }
}
```

A MIXED MODE EXPRESSION is an expression that contains components that are of different data types.

In this example, the expression q equals w divided by z is a **uniform-mode** expression because the types of the components on both the left and the right are all the same…they are all floating point types.

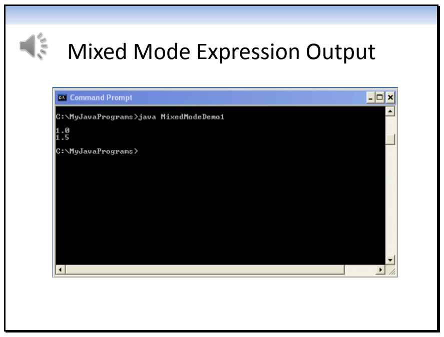
The remaining two expressions involving z are mixed-mode expressions because they contain a mixture of floating point types and integer types.

Before we go any further, I'd like you to take a minute right now and pause this lecture.

Then, I'd like you to write down what you think the two outputs of this program will be when it runs.

When you finish writing down your answers, go ahead and continue the lecture.

3



Mixed Mode Expression Output

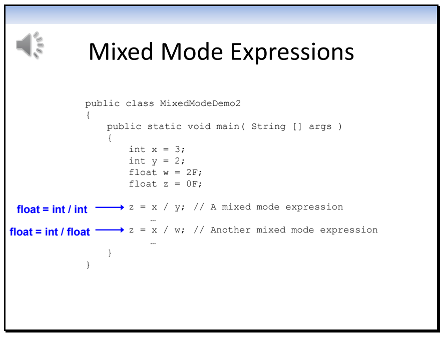
```
C:\MyJavaPrograms>java MixedModeDemo1

1.0
1.5

C:\MyJavaPrograms>
```

Here's the output of the MixedModeDemo1 program.

The first output shouldn't be much of a surprise. In the program, the variables x and y were both integers, and we've already learned that when integers are divided any remainder is lost. X had the value 3 and y had the value 2, so integer division makes the result 1 with no remainder.

For the second output, the integer x was divided by a floating point variable with a value of 2. Since one of the variables was a float, Java gave the result of the division a floating point type.

4



Mixed Mode Expressions

```
public class MixedModeDemo2
{
    public static void main( String [] args )
    {
        int x = 3;
        int y = 2;
        float w = 2F;
        float z = 0F;

        z = x / y;  // A mixed mode expression
            …
        z = x / w;  // Another mixed mode expression
            …
    }
}
```

float = int / int
float = int / float

Let me explain how Java handles mixed mode expressions.

In this program the statement z equals x divided by y is a mixed mode expression. Both x and y are declared as integer types, but z is declared as a floating point type. So, we essentially have the result of an integer/integer division being assigned to a floating point type.

When the right-hand division is performed, since both x and y are integers, the result is an integer. Since the left-hand side, z, is a float the types don't match.

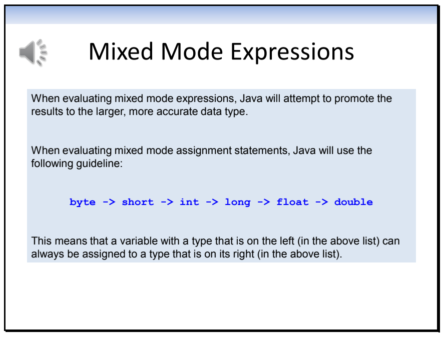
In these mixed-mode situations, Java has to decide how to handle things, because a variable can only have one type. The way Java handles situations like this is to provide what is called an automatic type conversion. Simply put, Java attempts to convert the result of an expression or assignment to the largest or more accurate data type.

In our example, since a float is larger than an integer, it converts the right hand expression to a float and makes the assignment. The result of the division expression x divided by y is an integer 1…because remainders are truncated when integers are divided. The integer result

1 is then converted to a float and assigned to z.

The second mixed-mode expression, z equals x divided by w, has an integer / float expression on the right-hand side and a float expression on the left-hand side. When Java evaluates the right-hand expression, it promotes the result to a float, since the largest, more accurate type in that expression is a float. Then, it assigns the floating point result to z.

5



Mixed Mode Expressions

When evaluating mixed mode expressions, Java will attempt to promote the results to the larger, more accurate data type.

When evaluating mixed mode assignment statements, Java will use the following guideline:

```
byte -> short -> int -> long -> float -> double
```

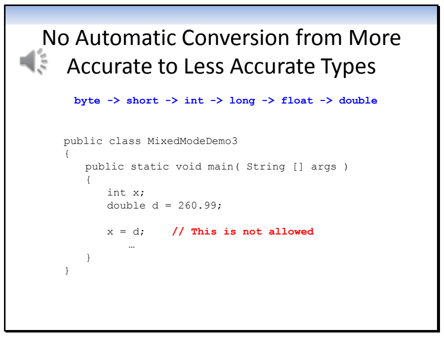
This means that a variable with a type that is on the left (in the above list) can always be assigned to a type that is on its right (in the above list).

In the last example we saw how Java deals with mixed-mode expressions. The basic rules are summarized in general form here.

When evaluating mixed mode expressions, like expressions on the right-hand side of an assignment, Java will attempt to promote each sub-expression to the larger, more accurate type.

When evaluating mixed mode assignment statements, Java will use the illustrated guideline. So, for example, a byte type can always be assigned to a short, an int, a long, a float, or a double. An int can always be assigned to a long, a float or a double…and so forth.

6



No Automatic Conversion from More Accurate to Less Accurate Types

```
byte -> short -> int -> long -> float -> double
```

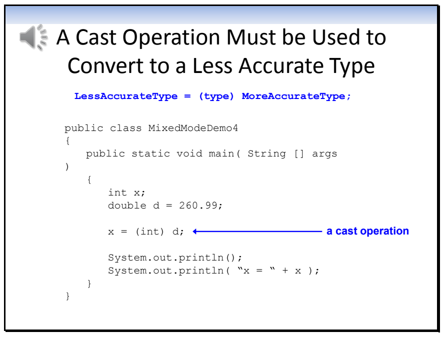
```
public class MixedModeDemo3
{
   public static void main( String [] args )
   {
      int x;
      double d = 260.99;

      x = d;    // This is not allowed
        …
   }
}
```

Java does not allow automatic type conversion from more accurate types to less accurate types.

It will not allow a variable of a type that is to the right of another on the above list to a type that is on its left. For example, an int cannot be assigned to a short or a byte. A float cannot be assigned to a long or a double, and so forth.

If such an assignment is attempted, the program will not compile. In this program, for example, there is an attempt to convert a double precision variable, d, to an integer type…and the program will not compile.

7



A Cast Operation Must be Used to Convert to a Less Accurate Type

```
LessAccurateType = (type) MoreAccurateType;
```

```
public class MixedModeDemo4
{
   public static void main( String [] args
)
   {
      int x;
      double d = 260.99;

      x = (int) d;

      System.out.println();
      System.out.println( "x = " + x );
   }
}
```

a cast operation

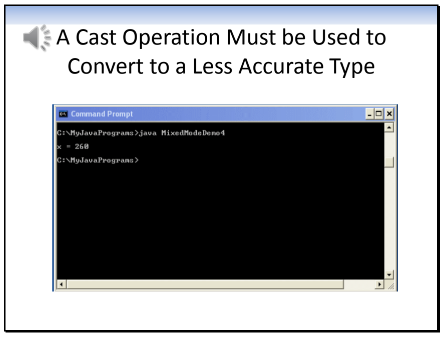
We've learned that Java does not allow automatic type conversion from more accurate types to less accurate types.

More accurate to less accurate conversions can be done, but you must use something called a **cast operation** to assign a more accurate type to a less accurate type. The general syntax for a cast operation is illustrated here. You put the type you are casting to inside a set of parentheses.

This is a slightly modified version of the last example, the one that wouldn't compile. In this program, the double precision variable d is cast to an integer type before the assignment is made.

Let's see what the value of x will be.

8

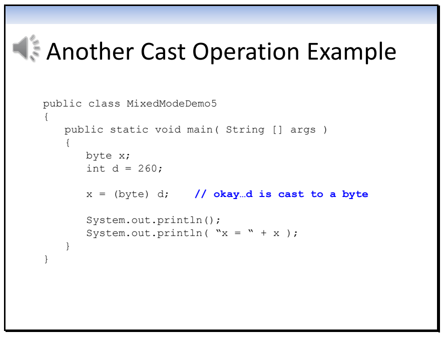


A Cast Operation Must be Used to Convert to a Less Accurate Type

```
C:\MyJavaPrograms>java MixedModeDemo4
x = 260
C:\MyJavaPrograms>
```

In this case the fractional part of d was thrown away, and the whole number part of d was assigned to x, so x is 260.

9



Another Cast Operation Example

```
public class MixedModeDemo5
{
   public static void main( String [] args )
   {
      byte x;
      int d = 260;

      x = (byte) d;    // okay…d is cast to a byte

      System.out.println();
      System.out.println( “x = “ + x );
   }
}
```

Here’s another example, In this example, let’s try to cast an integer to a byte. I’ll use the same integer value as in the last example…260.

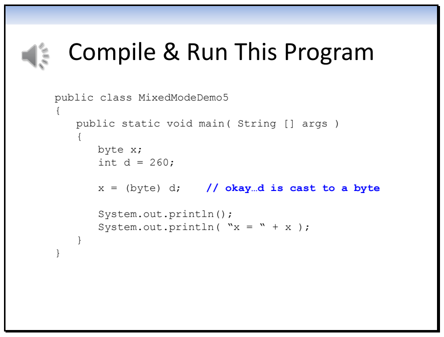
This program will compile and run.

Now, at this point I’d like you to pause the lecture.

Once you’ve paused the lecture, I’d like you to take a close look at the program. Can you guess what the program output will be?

Go ahead and write down what you think the output will be. When you’re done, continue the lecture.

10



Compile & Run This Program

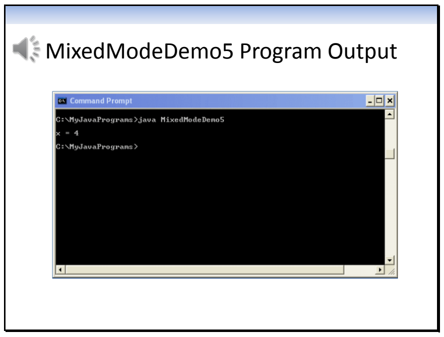
```
public class MixedModeDemo5
{
    public static void main( String [] args )
    {
        byte x;
        int d = 260;

        x = (byte) d;    // okay…d is cast to a byte

        System.out.println();
        System.out.println( "x = " + x );
    }
}
```

Please pause the lecture one more time.

I'd like you to compile and run this program, and compare its output to what you guessed it would be.

When you've run the program, continue the lecture so that we can discuss the output.

11



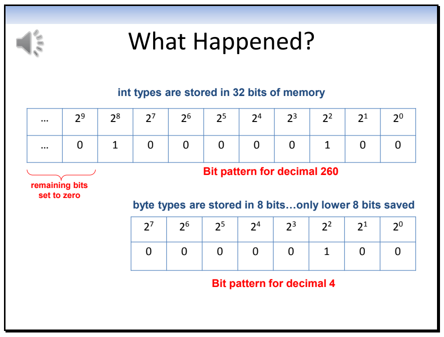
MixedModeDemo5 Program Output

```
C:\MyJavaPrograms>java MixedModeDemo5
x = 4
C:\MyJavaPrograms>
```

Hmm…the output is 4. How on earth did that happen?

I'll explain in a minute, but before I do I want to make a point of saying that you have to be very careful when using casting operations…and it helps to know exactly what goes on inside the computer when you use these operations.

12



What Happened?

int types are stored in 32 bits of memory

…	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
…	0	1	0	0	0	0	0	1	0	0

remaining bits set to zero

Bit pattern for decimal 260

byte types are stored in 8 bits…only lower 8 bits saved

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	1	0	0

Bit pattern for decimal 4

To understand what happened in the last example, we need to get inside the computer a little more. In the first lesson of this course, we introduced some basic computing concepts. We learned that computers translate everything they work with into zeroes and ones…both programming instructions and data.

When we declare an integer variable in a program and assign it the decimal value 260, the computer stores this value in memory in the form of a binary number consisting of 0's and 1's.
Java int types are stored in 4 bytes or 32 bits of memory. So, in memory the decimal value of 260 looks like this [*].

When we look at the binary equivalent of a decimal number we refer to it as a bit pattern. Each bit corresponds to a power of 2. Note that the ninth bit, corresponding to 2 to the 8th power is set to 1, and the 3rd bit, corresponding to 2 to the 2nd power is set to one. All the other bits are set to zero.

To get the decimal equivalent of a binary number we look at the bits that are set to one and add the total values…so, 2 to the 8th power is 256, 2 to the 2nd power is 4…making the decimal equivalent 260.

You may recall from an earlier lecture that byte types are always stored in 1 byte…or 8 bits…of memory. The maximum decimal value that can be represented in 8 bits of memory is 128. When the integer variable was cast to a byte type in the last program, only the first 8 bits were kept…the rest were thrown away. So, the resulting bit pattern after the cast looks like this [*]. The decimal equivalent of this bit pattern is 4.