1.11 Notes

## Casting Your Type

It's sometimes puzzling that a small number like -32 uses the same storage space as a big number like 2,147,483,647, but it is easier to have a few constant units of storage than one for every possible value! Two bytes of storage is sufficient to hold any **int**, big or small.

Java reserves enough memory storage for each variable's declared type. Consequently, the type of a variable must be declared before it can be used.

int anIntegerValue = 1.6180339887; < --- invalid   
double aDoubleValue = 365; < --- valid

Still, sometimes a programmer just has to be able to use a double value as an int, or an int as a double. Fortunately, Java provides several simple ways to handle these situations and avoid two common types of errors.

**Syntax error:** The program violates a rule of the language.

**Design error**: The program compiles and runs, but the output is incorrect. Also known as a run-time error.

Syntax errors occur in programs all the time; if you violate Java's rules, the compiler will be the first to let you know about it. Many errors can be traced to simple typographical mistakes as well as incorrectly written arithmetic expressions. Calculations involving mixed data types (e.g. **int**s and **double**s) are particularly troublesome, due to memory storage issues.

The memory requirements and corresponding range of values for each primitive data type are shown in the following table:

| **Part 1 Primitive Data Type** | | |
| --- | --- | --- |
| **Primitive Data Type** | **Size (bytes)** | **Range** |
| boolean | 1 | true or false |
| char | 2 | Unicode character set |
| byte | 1 | -27 to 27-1 (-128 to 127) |
| short | 2 | -215 to 215-1 (-32768 to 32767) |
| int | 4 | -231 to 231-1 (-2147483648 to 2147483647) |
| long | 8 | -263 to 263-1 |
| float | 4 | -3.4 x1038 to 3.4 x 1038 |
| double | 8 | -1.8 x 10308 to 1.8 x 10308 |

As you can see, **double**s are more precise and require more storage space (bytes) than **int**s. When a **double** is converted to an **int**, it has less precision and requires less memory … a narrowing of the type. In contrast, it is a widening of type when an **int** is converted to a **double**.

Java provides three conversion techniques to avoid syntax errors in mixed data type arithmetic expressions.

**Assignment Conversion** occurs when a value with less precision is assigned to a variable with greater precision. For example:

double x = 100;

In this case, assigning an integer value to a double variable is a widening conversion that causes 100 to be automatically converted to 100.0.

However, attempting to assign a double to an int (a narrowing conversion), as in the following example, is not possible.

int z = 100.0;

The compiler will issue a possible loss of precision error in this situation.

**Arithmetic Promotion** also happens automatically in an expression of mixed types. For example:

double x = 3.14 \* 10 \* 10;

The 10s in this expression are ints, but 3.14 is a double. In order to calculate the answer, a widening conversion occurs, automatically promoting the 10s to doubles. The data type of the result of the expression is a double. After this statement is executed, x contains a value of 314.0, a decimal number.

**Casting** allows promotion and demotion of types, but it must be done explicitly. For example:

int x = (int)3.14 \* 10 \* 10;

Placing the int type in parentheses in front of 3.14 (a double) causes the compiler to treat 3.14 as if it were just 3 (an int). This cast is a narrowing conversion, so after this statement is executed, x contains a value of 300, an integer. It is important to note, however, that casting does not actually change the number 3.14 in any way; it simply causes the program to treat the value as if it were actually a 3 for the purposes of this calculation only.

Generally, there is no problem with a widening conversion (e.g., an int to a double), but narrowing conversions (e.g. a double to an int) cause a loss of information and should be avoided unless it is intentionally done.

Calculating mistakes are not always caught by the compiler, as the following example illustrates:

double average = 4 + 5 + 3 + 3 + 4 / 5;

System.out.println("The GPA is: " + average);

Without knowing anything about the program in which these statements appear, it is clear that the intent is to calculate a student's GPA; however, the following output is generated:

**screen shot of the BlueJ terminal window with the output "The GPA is: 15.0"**

You might wish your GPA was this high, but it is obvious that something is wrong! There are four possible sources of error in this situation. Look at them in this order:

1. Did you print the right variable? Always check to see if the variable you intended to print is the one that you actually typed in the print statement. Printing the wrong variable is a common source of output error, and it is quick and easy to fix. (Is that the problem in this example?)
2. Is your data value correct? Make sure you typed in the right numbers. Lots of mistakes occur due to using the wrong numbers. (Are the five values correct?)
3. Is the denominator correct? It's an average, so if you don't divide by the same number of values being added, the answer will be incorrect. (Is there a 5 in the denominator?)
4. Is the numerator correct? The values (from Step 3) were correct, so check to see if they are being added correctly. (Is the numerator that will be divided by the denominator 19?)

The last step identifies the problem. The computer followed the order of operations and evaluated the arithmetic expression as it was written, but the programmer forgot to put parentheses around the five numbers being added, to force that quantity to be used as the numerator. Consequently, only the number 4 is being divided by 5! Don't be stung by neglecting the operator precedence rules.

Errors in mixed type calculations are easily solved with implicit or explicit type conversions. Arithmetic conversion and promotion automatically take place, but casting must be specifically implemented.

The best way to understand the implications of mixed type calculations is to see some examples of implicit (i.e., arithmetic conversion and arithmetic promotion) as well as explicit (i.e., casting) primitive data type conversions. As always, you should try modifying the practice programs and adding your own statements to check your understanding of this important concept.