Expressions, Data Conversion, and Input

- Expressions
- Operators and Precedence
- Assignment Operators
- Data Conversion
- Input and the Scanner Class
- Reading for this class: L&L, 2.4-2.6, App D
Expressions

• An *expression* is a combination of one or more operators and operands

• *Arithmetic expressions* compute numeric results and make use of the arithmetic operators:

  - Addition: +
  - Subtraction: -
  - Multiplication: *
  - Division: /
  - Remainder: %

• If either or both operands used by an arithmetic operator are floating point, then the result is a floating point
Division and Remainder

- If both operands to the division operator (/) are integers, the result is an integer (the fractional part is discarded)
  
  \[
  14 \div 3 \quad \text{equals} \quad 4 \\
  8 \div 12 \quad \text{equals} \quad 0
  \]

- The remainder operator (%) returns the remainder after dividing the second operand into the first
  
  \[
  14 \% 3 \quad \text{equals} \quad 2 \\
  8 \% 12 \quad \text{equals} \quad 8
  \]
Operator Precedence

• Operators can be combined into complex expressions

\[
\text{result} = \text{total} + \text{count} / \text{max} - \text{offset};
\]

• Operators have a well-defined precedence which determines the order in which they are evaluated

• Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation

• Arithmetic operators with the same precedence are evaluated from left to right, but parentheses can be used to force the evaluation order

• See Appendix D for a more complete list of operators and their precedence.
Operator Precedence

• What is the order of evaluation in the following expressions?

\[
\begin{align*}
\text{a + b + c + d + e} & \quad \text{a + b * c - d / e} \\
1 & \quad 2 & \quad 3 & \quad 4 & \quad 3 & \quad 1 & \quad 4 & \quad 2 \\
\text{a / (b + c) - d % e} & \\
2 & \quad 1 & \quad 4 & \quad 3 \\
\text{a / (b * (c + (d - e)))} & \\
4 & \quad 3 & \quad 2 & \quad 1
\end{align*}
\]
Assignment Revisited

• The assignment operator has a lower precedence than the arithmetic operators

First the expression on the right hand side of the = operator is evaluated

\[
\text{answer} = \text{sum} / 4 + \text{MAX} * \text{lowest};
\]

Then the result is stored in the variable on the left hand side
Assignment Revisited

• The right and left hand sides of an assignment statement can contain the same variable

  First, one is added to the original value of count

  \[
  \text{count} = \text{count} + 1;
  \]

  Then the result is stored back into count (overwriting the original value)
Increment and Decrement

- The increment and decrement operators use only one operand
- The *increment operator* (++) adds one to its operand
- The *decrement operator* (--) subtracts one from its operand
- The statement
  
  \[ \text{count}++; \]

  is functionally equivalent to

  \[ \text{count} = \text{count} + 1; \]
Increment and Decrement

• The increment and decrement operators can be applied in:
  – postfix form:
    
    \[
    \text{count}++ \quad \text{count}--
    \]
  – prefix form:
    
    \[
    ++\text{count} \quad --\text{count}
    \]

• These operators update the value in the memory location

• When used as part of a larger expression, the prefix form adds or subtracts one BEFORE the rest of the expression is evaluated and the postfix form does it AFTERWARDS

• Because of these subtleties, the increment and decrement operators should be used with care
Assignment Operators

• Often we perform an operation on a variable, and then store the result back into that variable.

• Java provides assignment operators to simplify that process.

• For example, the statement

```java
num += count;
```

is equivalent to

```java
num = num + count;
```
Assignment Operators

There are many assignment operators in Java, including the following:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Equivalent To</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>x += y</td>
<td>x = x + y</td>
</tr>
<tr>
<td>-=</td>
<td>x -= y</td>
<td>x = x - y</td>
</tr>
<tr>
<td>*=</td>
<td>x *= y</td>
<td>x = x * y</td>
</tr>
<tr>
<td>/=</td>
<td>x /= y</td>
<td>x = x / y</td>
</tr>
<tr>
<td>%=</td>
<td>x %= y</td>
<td>x = x % y</td>
</tr>
</tbody>
</table>
Assignment Operators

• The right hand side of an assignment operator can be a complex expression

• The entire right-hand expression is evaluated first, then the result is combined with the original variable

• Therefore

\[
\text{result} /= (\text{total} - \text{MIN}) \% \text{num};
\]

is equivalent to

\[
\text{result} = \text{result} / ((\text{total} - \text{MIN}) \% \text{num});
\]
Assignment Operators

• The behavior of some assignment operators depends on the types of the operands

• If the operands to the += operator are strings, the assignment operator performs string concatenation

• The behavior of an assignment operator (+=) is always consistent with the behavior of the corresponding operator (+)
Data Conversion

- Sometimes it is convenient to convert data from one type to another.
- For example, in a particular situation we may want to treat an integer as a floating point value.
- These conversions do not change the type of a variable or the value that's stored in it – they only convert a value as part of a computation.
Data Conversion

• Conversions must be handled carefully to avoid losing information

• *Widening conversions* are safest because they tend to go from a small data type to a larger one (such as a *short* to an *int*)

• *Narrowing conversions* can lose information because they tend to go from a large data type to a smaller one (such as an *int* to a *short*)

• In Java, data conversions can occur in three ways:
  – assignment conversion
  – promotion
  – casting
Assignment Conversion

- *Assignment conversion* occurs when a value of one type is assigned to a variable of another.

- For example, the following assignment converts the value stored in the `dollars` variable to a `double` value:
  ```java
double money;
int dollars = 123;
money = dollars;    // money == 123.0
```

- Only widening conversions can happen via assignment.

- The type and value of `dollars` will not be changed.
Data Conversion

• *Promotion* happens automatically when operators in expressions convert their operands

• For example, if `sum` is a `double` and `count` is an `int`, the value of `count` is promoted to a floating point value to perform the following calculation:

  ```java
double result = sum / count;
```

• The value and type of `count` will not be changed
Casting

- *Casting* is a powerful and dangerous conversion technique.
- Both widening and narrowing conversions can be done by explicitly casting a value.
- To cast, the desired type is put in parentheses in front of the value being converted.
- For example, if `total` and `count` are integers, but we want a floating point result when dividing them, we cast `total` or `count` to a double for purposes of the calculation:

  ```java
  double result = (double) total / count;
  ```
- Then, the other variable will be promoted, but the value and type of `total` and `count` will not be changed.
Some Special Cases

• The default type of a constant with a decimal point is double:
  
  ```java
  float f = 1.2;  // narrowing conversion
  float f = (float) 1.2  // needs a cast
  ```

• Results of `int` divide by zero are different from `float` or `double` divide by zero

• If `int` `count == 0`, depends on type of sum:
  
  ```java
  ave = sum/count;  // if int, exception
  ave = sum/count;  // if double, "NaN"
  ```
Reading Input

- Programs generally need input on which to operate.
- The `Scanner` class provides convenient methods for reading input values of various types.
- A `Scanner` object can be set up to read input from various sources, including from the user typing the values on the keyboard.
- Keyboard input is represented by the `System.in` object.
Reading Input

• The following line allows you to use the standard library Scanner class in statements in your class:

   import java.util.Scanner;

• The following line creates a Scanner object that reads from the keyboard:

   Scanner scan = new Scanner(System.in);

• The `new` operator creates the `Scanner` object

• Once created, the `Scanner` object can be used to invoke various input methods, such as:

   String answer = scan.nextLine();
Reading Input

- The **Scanner** class is part of the **java.util** class library and must be imported into a program to be used.
- See **Echo.java** (page 89)
- The **nextLine** method reads all of the input until the end of the line is found.
- The details of object creation and class libraries are discussed later in the course.
Input Tokens

• Unless specified otherwise, *white space* is used to separate the elements (called *tokens*) of the input

• White space includes space characters, tabs, new line characters

• The `next` method of the `Scanner` class reads the next input token and returns it as a String

• Methods such as `nextInt` and `nextDouble` read data of particular types

• See [GasMileage.java](#) (page 90)