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

Operators and Operands

- Operand: Can be any element that has some value:
  - A literal: 1, -2.5, true, false, 'd', “Hello World”
  - A variable: name, balance, courseTitle
  - (The result of) a method call: student.getName()
- Operator: Something that computes a result using one or more operands:
  - 1 + 2, 6 / 3, !studentIsSenior, count++
  - 5 * 4 == 10 * 2, 18 – 6 != 6 - 18

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 (i.e., **decimal**), 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 / 3      equals      4
8 / 12      equals      0
```

- The remainder operator (%) returns the remainder after dividing the second operand into the first

```
14 % 3      equals      2
8 % 12      equals      8
```

Operator Precedence

- Operands and operators can be combined into **complex expressions**

```
result  =  total + count / max - 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?

```
a + b + c + d + e
  1   2   3   4

a + b * c - d / e
  3   1   4   2
```

Without parentheses:

```
a / b + c - d % e
  1   3   4   2
```

With parentheses:

```
a / (b + c) - d % e
  2    1    4   3
```

```
a / (b * (c + (d - e)))
  4    3    2    1
```

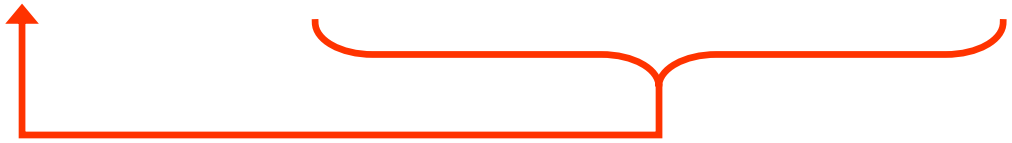
Assignment Revisited

- The assignment operator has a **lower** precedence than the arithmetic operators

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

```
answer  =  sum / 4 + MAX * lowest;
```

4 1 3 2



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

```
count  =  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

```
count++;
```

  is functionally equivalent to

```
count = count + 1;
```

Increment and Decrement

- The increment and decrement operators can be applied in:
  - *postfix form*:
    `count++`
      - `Gets current value, then adds 1`

    `count--`
      - `Gets current value, then subtracts 1`
  - *prefix form*:
    `++count`
      - `Adds 1 and then gets new value`

    `--count`
      - `Subtracts 1 and then gets new value`

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

```
num += count;
```

is equivalent to

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

Assignment Operators

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

Operator	Example	Equivalent To
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

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

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```

Expressions such as the former, if used correctly, can enhance your code's readability

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 decimal value
- These conversions **do not change** the type of a variable or the value that's stored in it – they only convert **the value itself** 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

```
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. `dollars` is still an int equal to 123 (not 123.0)

Promotion

- *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:

```
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:

```
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:

```
float f1 = 1.2;  // narrowing!
float f1 = (float) 1.2  // needs a cast
float f2 = 1.2f; // OR use a float literal
```

- The default type of a whole number is int
- This causes an odd behavior where a literal whole number value too large for an int will cause a compiler error:

```
long longVar = 3000000000; // wrong!
```

- Compiler tries to interpret it as an **`int`** - fails!
- Use a long literal – ex.: **`3000000000L`**

Some Special Cases

- Results of `byte, float, int,` or `long` divide by zero are different from `float` or `double` divide by zero
- If `int count` equals `0`, depends on type of sum:

```
ave = sum/count;// if int, exception
```

Throws an **exception** (i.e., crashes program) because it is mathematically impossible

```
ave = sum/count;// if double,
"Infinity"
```

The `float` and `double` types can have a value of "Infinity", unlike integer types

Character Arithmetic

- Because characters are associated with 16-bit integer values, you can do *arithmetic with characters*!
- For example, the expression `'b' - 'a'` will evaluate to `1` because the integer value of 'a' is one greater than that of 'b'.
- In addition, you can do arithmetic with characters and other numeric types, and the standard rules of data conversion (e.g. widening vs narrowing) will apply.

Character Arithmetic

- **Statements:** **Prints:**

```
System.out.println('a');                  // a
System.out.println(97);                   // 97
System.out.println((int) 'a');            // 97
System.out.println((char) 97);            // a
```

- **These five will . . . . . print as:**

```
int i = 0;
System.out.println((char) ('A' + i++));
System.out.println((char) ('A' + i++));
System.out.println((char) ('A' + i++));
System.out.println((char) ('A' + i++));
```

```
A
B
C
D
```

Character Arithmetic

- **Why does... print as?**

```
System.out.println('a');                    // a
```

Character literal: **'a'**

```
System.out.println(97);                     // 97
```

Integer literal: **97**

```
System.out.println((int) 'a');              // 97
```

Character value **converted** to an `int` value: **97**

```
System.out.println((char) 97);              // a
```

Integer value **converted** to a `char` value: **'a'**

Character Arithmetic

- **Why does...** **print as?**

```
int i = 0;
System.out.println((char) ('A' + i++));
System.out.println((char) ('A' + i++));
System.out.println((char) ('A' + i++));
System.out.println((char) ('A' + i++));
```

```
A
B
C
D
```

- It has to do with the **steps of conversion**:

1) `'A' + i++` → **char** value of `'A'` is promoted to **int**: 97
2) 97 + ***i*** → evaluates to an **int**
3) The cast converts the resulting **int** value to a **char** value, the latter of which is what gets printed.

(*NOTE: The letters are printed successively because* ***i*** *starts off as zero and gets post-incremented*)

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 (not available by default like String) 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)