Lecture 08: Operators

Sierra College
CSCI-12
Spring 2015
Mon 02/23/15

Announcements

- **General**
  - A bit behind on my grading, but working to get caught up
    - I am working to meet a 3/2 deadline for a distance learning review of an online version of THIS course for Fall 2015
- **Schedule**
  - A couple of slight assignment mods, see schedule items in **RED**
  - **Spring Pass/No-Pass deadline is Monday 3/2**
- **Past due assignments**
  - HW06: Template, accepted thru Weds 2/25 @ 11pm
- **Current assignments**
  - HW07: Variables, due Tues 2/24 @ 11pm
- **New assignments**
  - HW08: Operators, due Fri 2/27 @ 11pm (*lab time Weds*)
    - Use data such as declared last time, and perform calculations
    - Still using hardwired data: next time, we'll see how to get external input read in

Lecture Topics

- **Last time**:
  - DATA in Java
    - Variables, datatypes, literals, constants
- **Today**:
  - Tail end of last lecture: setting up data
  - Demo: using the jGRASP debugger to look at code
  - OPERATORS in Java

What Are Operators?

- Now that we can express program data using **variables** and **datatypes**… what do we do with that data??
- We use **operators** to write **expressions** which will "operate upon" (change, manipulate) data
  - Equations, formulas, algorithms, method calls, instructions, data manipulations, etc.
- **Operators** are special symbols in Java
  - Examples: **+ - * / =** and many others
- **Operands** are the data that they "operate upon"
  - Variables, constants, literals, objects, method calls, etc.

Operator Considerations

- The **number** of operands involved
  - **Unary operators**: **operator operand** (prefix form)
    operand operator (postfix form)
  - **Binary operators**: **operand operator operand**
  - Java also has one **ternary operator** (we'll get to it later…)
- The **datatypes** of the operands involved
  - Most often we have both two operands of the same type
  - But Java also provides for **mixed datatypes** used together
- The **order** in which operations get performed
  - This is known as **precedence**
- The **direction** of evaluation
  - Left-to-right, or right-to-left

Expressions

- An **expression** is some group of operators and operands that evaluates to one single value on the RHS
  - An **expression** is less than a **statement**
  - An expression cannot stand on its own and be executed
- The RHS **value** of the expression is assigned to a LHS **target** as part of a **statement**:

  target = expression;

- A **target** is an object or variable or constant, having a datatype compatible with the value of the expression
  - If target is a variable or object, its value is updated from its last value
  - If target is a constant, remember that its value can't change from its initial value, if the *final* keyword is used
- Examples on next slide…

Expression Examples

```
int  numberOfWins = 5;                  // expression is a literal
numberOfWins = 6;                       // target (variable) value gets updated

final int ONE_DOZEN = 12;               // use syntax for a constant
ONE_DOZEN = 13;                         // ILLEGAL: can't update value if final
final int BAKERS_DOZEN = ONE_DOZEN + 1;  // this is fine, though

double height = 5;
double width = 4;
double area = height * width;           // expression consists of two variables
                                        // and a binary operator

Person sam = new Person("Sam", "Smith");  // expression as an object creation
String lastName = sam.getLastName();      // expression as a method call
```

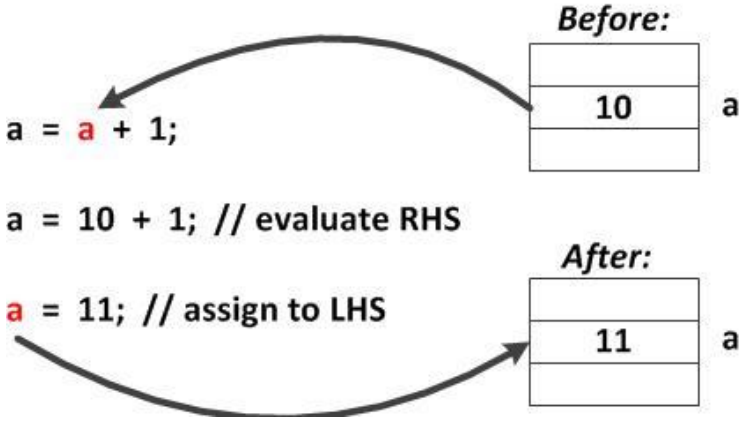
"Smith"

Assignment Operator

- The **assignment operator** is the "equals" symbol: **=**
- It is used to assign an expression value to a **target** (a variable or an object):

  target = expression;
- Think of the "=" symbol as a left-facing arrow:

  target ← expression
- **Assignment statements** are evaluated right-to-left:
  - First, the RHS is fully evaluated
  - Then, the LHS target receives the RHS value
    - a=a+1 would be impossible in algebra
    - But it is fine (and quite common) in programming
    - It's nothing but an update of a variable or object



Assignment Examples:

```
int numPlayers = 10;          // numPlayers initialized to 10
numPlayers = 8;               // numPlayers is updated

int legalAge = 18;            // legalAge initialized to 18
int voterAge = legalAge;      // voterAge updated using another variable
```

The next statement is illegal:

```
int length = width * 2;       // width is not yet defined
int width = 20;
```

It generates the following compiler error:

```
cannot find symbol            (although the message might not tell you which symbol…)
```

Arithmetic Operators

- Java's **arithmetic operators** are used to perform calculations on numeric data
  - All 4 usual operations, plus one new one (modulus)
  - All are binary operators (two operands, one on each side)

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder after division)

- See **OperatorsArithmetic.java** in **Example Source Code**

Integer Division/Modulus

- If we are dividing one integer by another integer:
  - **The division operator /** gives us the quotient after integer division: 18 / 16 = 1
  - The **modulus operator %** gives us the remainder after integer division 18 % 16 = 2
- Some sample uses:
  - Is a number even or odd, or evenly divisible by N?
  - Filling some container evenly, how much is left over?
  - How many of something per hour/day/week/month/year?
  - Number system conversions: how many times evenly does a base value (2, 10, 16) “go into” some number?

Integer Division/Modulus Examples

- When performing integer division:

  int a = 57;

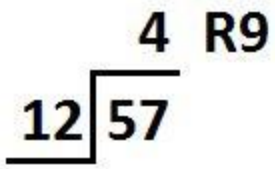
  int b = 12;

- / gives the normal integer division quotient

  a / b results in 4

- % gives the remainder after integer division

  a % b results in 9



- See **OperatorsIntegerDivision.java** in **Example Source Code**

Shortcut Operators

- Java provides two **shortcut operators: ++** and **--**
  - Shorthand for common increment/decrement by 1
  - There is NO SPACE between the operator characters
  - Each is a unary operator (one argument only)
  - Each comes in **prefix** and **postfix** versions

Operator	Example	Interpretation
++ (prefix)	++var	var = var + 1; use var in expression
++ (postfix)	var++	use var in expression var = var + 1;
-- (prefix)	--var	var = var – 1; use var in expression
-- (postfix)	var--	use var in expression var = var - 1;

- See **OperatorsShortcut.java** in **Example Source Code**

Shortcut Arithmetic Operators

- In addition to the preceding special cases, Java also provides more general shortcut operators
  - Again, there is NO SPACE between the operator characters
  - Order matters, the = sign is always second
  - Each is now a binary operator (variable, increment)

Shortcut Operator	Example	Equivalent Statement
+=	a += 3;	a = a + 3;
-=	a -= 10;	a = a – 10;
*=	a *= 4;	a = a* 4;
/=	a /= 7;	a = a / 7;
%=	a %= 10;	a = a % 10;

- See **OperatorsArithmeticShortcut.java** in **Example Source Code**

Operators Across Languages

language	Java	C	C++	Java Script	PHP	VB.NET
assignment	=	=	=	=	=	=
addition	+	+	+	+	+	+
subtraction	-	-	-	-	-	-
multiplication	*	*	*	*	*	*
division	/	/	/	/	/	/ usual \ integer
modulus	%	%	%	%	%	Mod
shortcut	++, --	++, --	++, --	++, --	++, --	(none)
arithmetic shortcut	+=, -=, *=, /=, %=	+=, -=, *=, /=, %=	+=, -=, *=, /=, %=	+=, -=, *=, /=, %=	+=, -=, *=, /=, %=	+=, -=, *=, /=, \=

Precedence

- **Precedence** is the determination of which operations get evaluated before other operations
  - Uses the rules of **operator precedence**
- Precedence in Java is similar to the rules of precedence you may recall from math or algebra
  - PEMDAS → "Please Excuse My Dear Aunt Sally"
  - Parentheses, exponentiation, multiplication, division, addition, subtraction
- However, Java has a more operators, so a more extensive list of precedence rules
  - A reasonably full list appears in Appendix B of your textbook
  - A complete list appears on the Oracle Java Trail website:
    http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html
    - It is also linked in the current Canvas lecture module

Precedence Example 1

- You have 2 quarters, 3 dimes, and 2 nickels.
- How many pennies are these coins worth?

```
int pennies = 2 * 25 + 3 * 10 + 2 * 5;
            =  50    +  30    +  10
            =  90
```

- Notes
  - * has higher precedence than +
    - So the multiplications are executed first, left to right
    - Then the additions are executed, left to right
  - We generally NEVER want to hardwire numbers into code like this; we always prefer to work symbolically
    - See **OperatorsPrecedence1.java** in **Example Source Code**

Operator Precedence

Operator	Order of same statement evaluation	Operation
( )	left - right	parenthesis for explicit grouping
++ --	right - left	post-increment, post-decrement
++ --	right - left	pre-increment, pre-decrement
* / %	left - right	multiplication, division, modulus
+ -	left - right	addition or *String* concatenation, subtraction
= += -= *= /= %=	right - left	Assignment, shortcut operators

Some Rules of Precedence

- Expressions in parentheses are evaluated first
- Operators are evaluated in the row order they appear in the precedence table, from top to bottom
- Operators in the same row have equal precedence, and are evaluated in the order in which they appear in an expression
- Assignment is always performed last, and is evaluated right to left
  - RHS is fully evaluated
  - RHS value is assigned to LHS variable/constant
  - LHS ← RHS (LHS "gets" whatever the HS evaluates to)

Precedence Example 2

- Translate $\frac{x}{2y}$ into Java: assume x=100, y=25

```
// This is incorrect!
double resultBad = x / 2 * y;        // 100/2*25 = 1250
```

=> $\frac{x}{2} * y$

```
// This is correct
// Parentheses force evaluation of 2 * y first
double resultGood = x / ( 2 * y );      // 2, correct
```

See **OperatorsPrecedence2.java** in **Example Source Code**

General Guidance on Precedence

- **Understand how it works**
  - You will encounter code that uses it
  - You need to understand how to decipher a program's intentions

- **But, don't rely upon it** in any new code you write
  - Be clear and explicit about your intent, with parentheses
  - Don't force every other user to figure out your intentions
  - Since you know what you mean, say so in your code!
    - With parentheses, and with comments if needed

Mixing Variable Datatypes

- Sometimes, all the variables in an expression are of the same datatype
- But other times, variable assignment is made using mixed datatypes
- Generally, this is allowed, as long as the LHS variable can accommodate the RHS datatype(s)
  - Smaller variables can be assigned to larger ones
  - Larger variables CANNOT be assigned to smaller ones
- Also, avoid the following situations (compiler errors):
  - Don’t redeclare a variable after it’s already been declared
  - Don’t try to change a datatype once it’s been assigned

Mixed-Type Expressions

- Arises when operands have differing datatypes
  - Avoid if reasonably possible
  - Plan out your datatypes beforehand
- Resolving this requires **type casting**
- Two types:
  - **Implicit type casting**
    - Done auto-magically by the compiler
  - **Explicit type casting**
    - Specified by the software developer

      (dataType) (expression)

Implicit Type Casting

- For any given operator, the compiler looks at both operands of an operator (for binary operators)
  - "Look left, look right"
- The lower precision operand is auto-promoted to the higher precision datatype
  - The rules of promotion are followed
  - Temporary: the datatype of any promoted operand is not permanently changed, just for purpose of expression evaluation
- The operation is performed
- Next operation, or else final assignment, is performed

Implicit Casting Example 1

- Valid assignment: lower precision into higher precision
  - *double* datatype is higher precision than *float* datatype
  - implicit casting is done (successfully)

```
float taxRate = 0.05f;
double salesTax = taxRate;
```

- Invalid assignment: higher precision into lower precision
  - *float* datatype is lower precision than *double* datatype
  - implicit casting is done (and fails, due to a compiler error)

```
double taxRate = 0.05;
float salesTax = taxRate;
```

- See **OperatorsCasting.java** in **Example Source Code**

Implicit Casting Example 2

- 2 variables are knowns
  - So declare + initialize them
- 1 variable is an output
  - So declare it only
  - It gets calculated later
- All 3 variables in line 28 calculation are of **mixed type**
  - RHS: (float * int) becomes float, via **implicit cast**
  - LHS: double ← float is OK
- Notice the output format
  - Due to internal numeric storage
  - We'll address this later…

```
14 public class OperatorsMixedTypes {
15
16     public static void main(String [] args) {
17
18         // data declaration/initialization
19         int qty = 10;
20         float retailPrice = 2.99F;
21         double salePrice;
22
23         // calculations
24
25         //   the following implicit casting takes place
26         //   1) float x int --> float (RHS)
27         //   2) double (LHS) <-- float (RHS)
28         salePrice = retailPrice * qty;
29
30         // outputs
31         System.out.println("The sale price of " + qty +
32                            " items which retail for $" + retailPrice +
33                            " is:\n$" + salePrice);
34
35     } // end main
36
37 } // end class
```

```
----jGRASP exec: java OperatorsMixedTypes

The sale price of 10 items which retail for $2.99 is:
$29.899999618530273

----jGRASP: operation complete.
```

- See ***OperatorsMixedTypes.java*** in **Example Source Code**

Compatible Data Types

A variable of any type in right column can be assigned to a variable of any type in left column

A smaller datatype can be assigned to a larger datatype, but not the other way around

Data Type	**Compatible Data Types**
byte	*byte*
short	*byte, short*
int	*byte, short, int, char*
long	*byte, short, int, long, char*
float	*float, byte, short, int, long, char*
double	*float, double, byte, short, int, long, char*
boolean	*boolean*
char	*char*

Rules of Promotion

The compiler applies the first of these rules that fits:

1. If either operand is a *double*, the other operand is converted to a *double*.
2. If either operand is a *float,* the other operand is converted to a *float.*
3. If either operand is a *long*, the other operand is converted to a *long*.
4. If either operand is an *int*, the other operand is promoted to an *int*
5. If neither operand is a *double*, *float, long,* or an *int*, both operands are promoted to *int*.
6. So any calculations involving *short* or *byte* types will get promoted to a minimum of *int* type.

Explicit Type Casting

- We can **explicitly** specify a recast of an operand or expression to a desired datatype:

  (dataType) (expression)

  Note: the parentheses around expression are optional if it is just one single variable

- Often see in in the context of **averaging** integer values

Explicit Casting Examples: Averaging

- Calculate the average of some scores:

```
int sumScores, countScores;   // assume these can be determined
double avgScore;              // desired result has decimal accuracy

// incorrect: RHS is performed as integer division
avgScore = sumScores / countScores;

// incorrect: RHS avg is already int result, too late to cast
avgScore = (double) (sumScores / countScores);

// correct: one RHS double term forces floating-point evaluation
avgScore = (double) sumScores / countScores;
avgScore = sumScores / (double) countScores;
```

- See **OperatorsCasting.java** in **Example Source Code**

For Next Time

- **Lecture Prep**
  - Text readings and lecture notes
- **Assignments**
  - See slide 2 for new/current/past due assignments