Eric Roberts
CS 106A

Handout #15
January 13, 2010

Expressions

Expressions

Eric Roberts
CS 106A
January 13, 2010

Holism *vs.* Reductionism

In his Pulitzer-prizewinning book, computer scientist Douglas Hofstadter identifies two concepts—***holism*** and ***reductionism***—that turn out to be important as you begin to learn about programming.

Hofstadter explains these concepts using a dialogue in the style of Lewis Carroll:

Achilles:	I will be glad to indulge both of you, if you will first oblige me, by telling me the meaning of these strange expressions, "holism" and "reductionism".
Crab:	Holism is the most natural thing in the world to grasp. It's simply the belief that "the whole is greater than the sum of its parts". No one in his right mind could reject holism.
Anteater:	Reductionism is the most natural thing in the world to grasp. It's simply the belief that "a whole can be understood completely if you understand its parts, and the nature of their 'sum'". No one in her left brain could reject reductionism.

The `Add2Integers` Program

```
class Add2Integers extends ConsoleProgram {
   public void run() {
      println("This program adds two numbers.");
      int n1 = readInt("Enter n1: ");
      int n2 = readInt("Enter n2: ");
      int total = n1 + n2;
      println("The total is " + total + ".");
   }
}
```

n1	n2	total
17	25	42

```
Add2Integers
This program adds two numbers.
Enter n1: 17
Enter n2: 25
The total is 42.
```

Expressions in Java

- The heart of the `Add2Integers` program from Chapter 2 is the line

  ```
  int total = n1 + n2;
  ```

  that performs the actual addition.
- The `n1 + n2` that appears to the right of the equal sign is an example of an ***expression***, which specifies the operations involved in the computation.
- An expression in Java consists of ***terms*** joined together by ***operators***.
- Each term must be one of the following:
  - A constant (such as `3.14159265` or `"hello, world"`)
  - A variable name (such as `n1`, `n2`, or `total`)
  - A method calls that returns a values (such as `readInt`)
  - An expression enclosed in parentheses

Primitive Data Types

- Although complex data values are represented using objects, Java defines a set of ***primitive types*** to represent simple data.
- Of the eight primitive types available in Java, the programs in this text use only the following four:

`int`	This type is used to represent integers, which are whole numbers such as 17 or –53.
`double`	This type is used to represent numbers that include a decimal fraction, such as 3.14159265.
`boolean`	This type represents a logical value (`true` or `false`).
`char`	This type represents a single character.

Constants and Variables

- The simplest terms that appear in expressions are ***constants*** and ***variables***. The value of a constant does not change during the course of a program. A variable is a placeholder for a value that can be updated as the program runs.
- A variable in Java is most easily envisioned as a box capable of storing a value.

  `total`

42

  (contains an `int`)
- Each variable has the following attributes:
  - A ***name***, which enables you to differentiate one variable from another.
  - A ***type***, which specifies what type of value the variable can contain.
  - A ***value***, which represents the current contents of the variable.
- The name and type of a variable are fixed. The value changes whenever you ***assign*** a new value to the variable.

Variable Declarations

- In Java, you must ***declare*** a variable before you can use it. The declaration establishes the name and type of the variable and, in most cases, specifies the initial value as well.
- The most common form of a variable declaration is

```
type name = value;
```

  where *type* is the name of a Java primitive type or class, *name* is an identifier that indicates the name of the variable, and *value* is an expression specifying the initial value.
- Most declarations appear as statements in the body of a method definition. Variables declared in this way are called ***local variables*** and are accessible only inside that method.
- Variables may also be declared as part of a class. These are called ***instance variables*** and are covered in Chapter 6.

Operators and Operands

- As in most languages, Java programs specify computation in the form of ***arithmetic expressions*** that closely resemble expressions in mathematics.
- The most common operators in Java are the ones that specify arithmetic computation:

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

- Operators in Java usually appear between two subexpressions, which are called its ***operands***. Operators that take two operands are called ***binary operators***.
- The `-` operator can also appear as a ***unary operator***, as in the expression `-x`, which denotes the negative of `x`.

Division and Type Casts

- Whenever you apply a binary operator to numeric values in Java, the result will be of type `int` if both operands are of type `int`, but will be a `double` if either operand is a `double`.
- This rule has important consequences in the case of division. For example, the expression

```
14 / 5
```

  seems as if it should have the value 2.8, but because both operands are of type `int`, Java computes an integer result by throwing away the fractional part. The result is therefore 2.
- If you want to obtain the mathematically correct result, you need to convert at least one operand to a `double`, as in

```
(double) 14 / 5
```

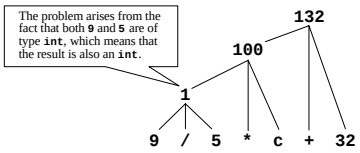
  The conversion is accomplished by means of a ***type cast***, which consists of a type name in parentheses.

The Pitfalls of Integer Division

Consider the following Java statements, which are intended to convert 100˚ Celsius temperature to its Fahrenheit equivalent:

```
double c = 100;
double f = 9 / 5 * c + 32;
```

The computation consists of evaluating the following expression:

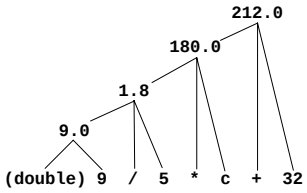


The Pitfalls of Integer Division

You can fix this problem by converting the fraction to a `double`, either by inserting decimal points or by using a type cast:

```
double c = 100;
double f = (double) 9 / 5 * c + 32;
```

The computation now looks like this:



The Remainder Operator

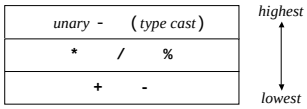
- The only arithmetic operator that has no direct mathematical counterpart is `%`, which applies only to integer operands and computes the remainder when the first divided by the second:

`14 % 5`	*returns*	`4`
`14 % 7`	*returns*	`0`
`7 % 14`	*returns*	`7`

- The result of the `%` operator make intuitive sense only if both operands are positive. The examples in the book do not depend on knowing how `%` works with negative numbers.
- The remainder operator turns out to be useful in a surprising number of programming applications and is well worth a bit of study.

Precedence

- If an expression contains more than one operator, Java uses ***precedence rules*** to determine the order of evaluation. The arithmetic operators have the following relative precedence:



  Thus, Java evaluates unary `-` operators and type casts first, then the operators `*`, `/`, and `%`, and then the operators `+` and `-`.
- Precedence applies only when two operands compete for the same operator. If the operators are independent, Java evaluates expressions from left to right.
- Parentheses may be used to change the order of operations.

Exercise: Precedence Evaluation

What is the value of the expression at the bottom of the screen?

```
( 1 + 2 ) % 3 * 4 + 5 * 6 / 7 * ( 8 % 9 ) + 10
```

Assignment Statements

- You can change the value of a variable in your program by using an ***assignment statement***, which has the general form:

  variable `=` *expression*`;`
- The effect of an assignment statement is to compute the value of the expression on the right side of the equal sign and assign that value to the variable that appears on the left. Thus, the assignment statement

```
total = total + value;
```

  adds together the current values of the variables `total` and `value` and then stores that sum back in the variable `total`.
- When you assign a new value to a variable, the old value of that variable is lost.

Shorthand Assignments

- Statements such as

```
total = total + value;
```

  are so common that Java allows the following shorthand form:

```
total += value;
```
- The general form of a ***shorthand assignment*** is

  variable *op*`=` *expression*`;`

  where *op* is any of Java's binary operators. The effect of this statement is the same as

  variable `=` *variable* *op* `(`*expression*`);`

  For example, the following statement multiplies `salary` by 2.

```
salary *= 2;
```

Increment and Decrement Operators

- Another important shorthand form that appears frequently in Java programs is the ***increment operator***, which is most commonly written immediately after a variable, like this:

```
x++;
```

  The effect of this statement is to add one to the value of `x`, which means that this statement is equivalent to

```
x += 1;
```

  or in an even longer form

```
x = x + 1;
```
- The `--` operator (which is called the ***decrement operator***) is similar but subtracts one instead of adding one.
- The `++` and `--` operators are more complicated than shown here, but it makes sense to defer the details until Chapter 11.

Extending `Add2Integers`

- The next few slides extend the `Add2Integers` program from Chapter 2 to create programs that add longer lists of integers. These slides illustrate three different strategies:
  - Adding new code to process each input value
  - Repeating the input cycle a predetermined number of times
  - Repeating the input cycle until the user enters a sentinel value

The **Add4Integers** Program

- You could easily change the **Add2Integers** into a program that added four integers just by adding additional variables, as shown in the following example:

```
public class Add4Integers extends ConsoleProgram {
   public void run() {
      println("This program adds four numbers.");
      int n1 = readInt("Enter n1: ");
      int n2 = readInt("Enter n2: ");
      int n3 = readInt("Enter n3: ");
      int n4 = readInt("Enter n4: ");
      int total = n1 + n2 + n3 + n4;
      println("The total is " + total + ".");
   }
}
```

- This strategy, however, is difficult to generalize and would clearly be cumbersome if you needed to add 100 values.

The Repeat-N-Times Idiom

One strategy for generalizing the addition program is to use the Repeat-N-Times idiom, which executes a set of statements a specified number of times. The general form of the idiom is

```
for (int i = 0; i < repetitions; i++) {
   statements to be repeated
}
```

The information about the number of repetitions is specified by the first line in the pattern, which is called the ***header line***.

The statements to be repeated are called the ***body*** of the **for** statement and are indented with respect to the header line.

A control statement that repeats a section of code is called a ***loop***.

Each execution of the body of a loop is called a ***cycle***.

The **AddNIntegers** Program

This program uses the Repeat-N-Times idiom to compute the sum of a predetermined number of integer values, specified by the named constant **N**.

```
public class AddNIntegers extends ConsoleProgram {
   public void run() {
      println("This program adds " + N + " numbers.");
      int total = 0;
      for (int i = 0; i < N; i++) {
         int value = readInt(" ? ");
         total += value;
      }
      println("The total is " + total + ".");
   }

   private static final int N = 100;
}
```

The Repeat-Until-Sentinel Idiom

A better approach for the addition program that works for any number of values is to use the Repeat-Until-Sentinel idiom, which executes a set of statements until the user enters a specific value called a ***sentinel*** to signal the end of the list:

```
while (true) {
   prompt user and read in a value
   if (value == sentinel) break;
   rest of loop body
}
```

You should choose a sentinel value that is not likely to occur in the input data. It also makes sense to define the sentinel as a named constant to make the sentinel value easy to change.

The **AddIntegerList** Program

This program uses the Repeat-Until-Sentinel idiom to add a list of integers, stopping when the user enters a value that matches the named constant **SENTINEL**.

```
public class AddIntegerList extends ConsoleProgram {
   public void run() {
      println("This program adds a list of integers.");
      println("Enter values, one per line, using " + SENTINEL);
      println("to signal the end of the list.");
      int total = 0;
      while (true) {
         int value = readInt(" ? ");
         if (value == SENTINEL) break;
         total += value;
      }
      println("The total is " + total + ".");
   }

   private static final int SENTINEL = 0;
}
```

Designing for Change

- While it is clearly necessary for you to write programs that the compiler can understand, good programmers are equally concerned with writing code that *people* can understand.
- The importance of human readability arises from the fact that programs must be maintained over their life cycle. Typically, as much as 90 percent of the programming effort comes *after* the initial release of a system.
- There are several useful techniques that you can use to make it easier for programmers who need to maintain your code:
  - Include comments to document your design decisions
  - Use names that express the purpose of variables and methods
  - Use indentation to make the structure of your programs clear
  - Use named constants to enhance readability and maintainability