CS112 Lecture: Working with Numbers

Last revised January 30, 2008

Objectives:

1. To introduce arithmetic operators and expressions
2. To expand on accessor methods
3. To expand on variables, declarations of variables, and assignment to variables
4. To introduce literal and symbolic constants
5. To introduce mechanisms for reading and writing numbers using class Text and standard IO for output and JOptionPane for input.
6. To introduce the generation of "random" numbers

Materials:

1. BlueJ Project with ScrollingSun, BankAccount, ICanCount, ICanCountConsoleApp, JOptionPaneDemo, Scribble

I. Operators and Expressions

A. One of the most basic and fundamental things a computer can do is to perform arithmetic - in fact, that's where the name "computer" comes from.

We've already seen one example of this:

ScrollingSun - computation of initial position and size for the sun as a function of the size of the canvas

SHOW CODE

DEMO with 100 x 100, 1000 x 700

B. In programming languages, numerical computations are typically specified by constructs known as *arithmetic expressions*. An arithmetic expression is constructed from two basic elements:

1. *operators* - which specify a particular computation (e.g. the operator + specifies the computation we know as addition).

2. *operands* - which are the things operated upon by the operators

   Example: in the expression 2 + 3, the 2 and 3 are operands and the + is the operator.

C. The set of permissible operators in Java is defined as part of the language, and includes both arithmetic operators and operators used for other purposes. For now, we consider only the arithmetic operators discussed in Bruce ch. 3 plus one other (there are many others).

+, -, *, /, % [not in Bruce - it stands for remainder]

1. Note that the symbol "-" actually represents two operators

   a) *Unary* minus take a single operand (as in - 3), and stands for negation

   b) *Binary* minus requires two operands (as in 3 - 2), and stands for subtraction

   (Actually, the symbol "+" also represents both a unary and a binary operator, though unary + doesn't do anything useful!)

2. Potential ambiguity in an expression is resolved by rules of *operator precedence*.

   a) Unary operators take precedence over binary operators

   b) *Multiplicative* binary operators (*, /, %) take precedence over other binary operators.

   c) The *additive* binary operators (+, -) are the lowest precedence operators of the ones we have looked at so far.

   d) Ties between operators at the same level are broken left to right for the operators we have considered so far. (There are some that break ties the other way!)

   Example: Order of evaluation of operators in

   ```
   - a + b % c / d - e * - f - g
   ```

   (Work out with class)

   ```
   - a + b % c / d - e * - f - g

   1   6   3   4   7   5 2   8
   ```

   equivalent to

   ```
   ( ( (- a) + ((b % c) / d) ) - (e * (- f)) ) - g
   ```

3. As noted in the above example, parentheses can be used to explicitly specify precedence - either because

   a) An order other than the normal one is needed

      Example: `(a + b) * (c + d)`

      or

   b) There is a desire to make the order clearer to the reader

D. Possible kinds of operands in Java expressions include:

1. Constants

2. Calls to accessor methods (e.g. `canvas.getWidth()` in the ScrollingSun example)

3. Variables (which we shall introduce shortly)

4. A parenthesized expression is actually treated as a single operand.

   Example: `2 * (1 + 2)`

II. Accessor Methods

A. Recall that, in our last lecture, we saw that behaviors of an object fall into four broad categories.

   What were they?

   ASK

1. Constructors - create new objects

2. Mutators - change the state of an object

3. Accessors - provide information about the state of an object without changing it

4. Destructors - specify what happens when an object is destroyed (not commonly used in Java)

B. Last tiime we looked at an example of a Java class that uses quite a few accessor methods of other objects (though defining none of its own). Let's look at it again

   PROJECT ScrollingSun

1. What are the uses of accessors of other objects that appear in this class?

   ASK

   getWidth() [4 times] and getHeight() - all on the canvas in the first line of begin()
   getWidth() [twice - on two different objects] in last line of begin()
   getWidth() on the canvas and getY() on a mouse position in the first line of onMouseDrag()
   getWidth [twice] and getHeight() on the canvas in the first line of onMouseExit()

2. Sometimes - as here - an accessor simply tells you the value of some component of an object's state (e.g its width, height, or position). It is a convention (though not a requirement) that such accessors have a name that begins with the word "get".

3. Other accessors may do a bit more computation, rather than simply giving a simple value. We saw several examples of such accessors in conjunction with robots. What were they?

   ASK

   facingNorth() [or South, East, West] - testing to see if a particular component of the state has a particular value
   anyBeepersInBeeperBag() - testing to see whether a particular component of the state is non-zero
   nextToABeeper() - similar

4. Sometimes, accessors may do quite a bit of computation

   Example: in a BankAccount class, one might have an accessor named computeInterest() which - in a sophisticated case - might involve computing an average balance for a period and then multiplying by the interest rate. Further, some accounts have rules like there is a minimum balance that must be maintained at all times to get interest at all - so the accessor would need to check the balance on each day to be sure it was always high enough.

C. Let's look at a previous example that <u>defined</u> some accessor methods: the BankAccount example we used on the first day of class

   PROJECT - Scroll down to bottom of class

1. What accessors are defined?

   ASK

   calculateInterest()
   reportBalance()
   getAccountNumber()

2. One of these simply provides access to an aspect of the state (an instance variable), while the other does a bit of computation based on the state, and one doesn't do any computation in this class, though it might be overridden to do so in a subclass. Which is which?

   ASK

3. Other than a name beginning with "get" (which pertains to just one of the methods), there is another "tip off" that indicates that a method is an accessor. What is it?

   ASK

   The return type of the method is something other than void. (Mutators are typically void)

   This is not a sure thing, though, since it is possible for a mutator to also return a value [often indicating whether the change was successful - e.g. a mutator that deletes a specific item from a list list may return true or false to indicate whether or not the item was actually there originally].

III. Variables

A. Methods in programs often must be able to manipulate various numeric quantities (e.g. the method that posts interest on a savings account must calculate the amount of interest by multiplying the current balance of the account times the interest rate.). We refer to such quantities within a program by using *variables*.

B. A variable is simply a symbolic name for a region of computer memory that can hold a piece of information needed during a computation. Four pieces of information characterize every variable:

   1. A name
   2. A type - what kind of value it can represent
   3. A storage allocation - a region of memory where its value is actually held
   4. A value.

C. Java requires that every variable used in a program be *declared* before it is used.

1. Of the characteristics listed above:

   The first three are fixed when the variable is declared.
   The fourth can change over time - hence the name variable.

2. A variable declaration consists of a type followed by one or more variables that are being declared (separated by commas) followed by a semicolon - e.g.

```
SimpleEscaperRobot karel;
int i;
```

3. So far, we have seen two places where variables can be declared. (There are actually two more kinds of variable declaration which we shall meet later.)

   ASK

   a) A variable that is declared inside a class but outside any method of the class (e.g. `sun` and `instructions` in class `ScrollingSun`) is called an instance variable.

      (1) Java would allow the declaration of these to occur either where they do, or between any two methods, or at the end between the last method and the closing brace. Usually, though, instance variables are only declared either at the very start of a class - just after the opening brace (as here) - or at the very end of a class - just before the closing brace.

      Where an instance variable is declared means nothing to Java, but may help the human reader of the program.

      (2) Every object constructed from a class has its own copy of each of the instance variables. Any method can access the instance variables of the particular object to which it is applied.

      Ex: In ScrollingSun, all three methods use both instance variables.

      (3) Instance variables also have a visibility specifier. Generally, this is the reserved word `private`, which means that the variable can only be accessed inside the class - e.g.

```
class Foo
{
     private int bar;
     ...
}

Foo f = new Foo();
f.bar <-- not allowed
```

This is an example of an important OO concept known as encapsulation.

b) We have also seen that a formal paremeter is a variable that is declared in a method header. Such a declaration only applies to that one method.

4. An instance variable declaration may optionally include initialization for the value of the variable - e.g.

```
int i = 42;
```

If an instance variable is not initialized when it is declared, it is given a default value - e.g. 0 for an integer.

5. Once a given name has been declared as the name of a variable, it cannot be used for a different variable in the same scope.

a) That is, a class cannot contain two instance variables with the same name.

b) A method cannot contain two parameters with the same name.

c) It is possible for a method to have a parameter with the same name as an instance variable of the class to which it belongs. In this case, within that one method the use of the name refers to the parameter, not the instance variable.

6. The requirement that variables be declared stands in contrast to some languages (e.g. JavaScript, BASIC) that allow *implicit declaration* of variables - i.e. simply *using* a name as a variable causes it to be declared.

D. The value of a variable is set or modified by an *assignment statement* which has the following form:

<variable> = <expression>

Examples:

```
x = 42;
y = x + 1;
x = x + 1;
```

E. It is important to note that a programming language variable differs in some significant ways from a mathematical variable.

1. A mathematical variable typically stands either for a fixed value that is currently *unknown* - as in

   $x^2 + 2x + 1 = 0$

   or for *all possible values* - as in the identity

   $x^2 + 2x + 1 = (x + 1)^2$

2. A programming language variable stands for something that can change over time.

3. The distinction is perhaps best illustrated by a statement like

   $x = x + 1$

   which is mathematically impossible but perfectly normal in a program (where it means "replace the current value of x by a new value that is one greater than its current value")

4. This distinction is related to a distinction between the mathematical and programming language meaning of " = ".

   a) In mathematics, "=" is *declarative* - it is an assertion that two things are equal, or perhaps is used in a context where we are asking whether two things are equal

   b) In programming languages like Java, "=" is *imperative* - it is an active verb that stands for assignment - i.e. it says "make this equal to".

   c) Because of the possibility of confusion between the mathematical and programming language meanings of "=", some languages (e.g. Pascal, Ada), use ":=" instead of "=" for assignment. (Java, however, uses "=" for assignment, with the expectation that the user will remember the difference between the "Java meaning" and the "mathematical meaning".)

IV. Constants

A. We have already indicated that one kind of operand that can appear in an expression is a constant.

B. Actually Java (and many programming languages) supports two kinds of constants.

1. A *literal constant* is one whose value is explicit.
2. A *symbolic constant* is a symbol that might otherwise look like a variable, but whose value has been fixed at the point where it is declared.

C. Java supports literal constants of various types

1. boolean: we have already met the two boolean constants, false and true
2. int: a string of decimal digits. It is also possible to represent int constants using octal or hexadecimal notation. (One warning: if the first digit of a multi-digit integer is 0, it is taken as octal, not decimal!)

3. There are a number of other types of constant which we will learn about later.

D. Symbolic constants are explicitly declared in the program

1. They are declared the same way as instance variables are, except that

a) The declaration includes the words `static final`.

b) The visibility may be either `private` or `public`, depending on whether or not the constant is meaningful outside the class.

c) The value must be specified at the point of declaration (since it can't be specified later!) The value can either be a literal, or an expression whose values are constants.

Example:

```
public static final int MINIMUM_AGE = 18;
```

2. It is standard practice (though not required by the compiler) that symbolic constants be given names that use all uppercase letters, with underscores used to separate words.

Note: the examples in the chapter in the Bruce book do not follow this convention, though most of the authors' examples do

3. Why should we use symbolic constants?

   a) Readability - a constant with a good name is more understandable than a "magic number" - especially it the magic number is either not readily recognized or could have several possible interpretations.

   b) Correctness - declaring a symbolic constant protects against accidentally mistyping a constant in one place if it is used several times. (If the user mistypes the name of a symbolic constant, the compiler will almost certainly catch the error)

   c) Maintainability - if the value needs to change, it's easier and more secure to change the constant declaration rather than each use. (Not an issue with constants like PI, but a likely issue with a constant like TAX_RATE).

E. As we shall see, a number of standard Java library classes define constants - e.g. the class `java.lang.Math` defines `Math.PI` and `Math.E`.

V. Output and Input of Numbers

A. Many programs follow the paradigm

   read some input
   do some computation
   produce some output

   We have spent most of our time discussing how we do computation. We now need to talk about how we can do input and output.

B. Unfortunately, this topic (especially input) is complicated by the fact that human users want to do input and output using strings of characters while computers represent numbers internally in binary form. Thus, outputting a number involves converting the internal binary form of a number to a string of characters, and inputting a number involves the reverse.

   1. Output is sometimes complicated by the need to control the format of the output - e.g. if printing a dollar amount, we want to use exactly two decimal places even if they are not strictly necessary to represent the value - e.g. as a dollar amount, we want to print 3 as 3.00.

2. Input is further complicated by the fact that the number the user enters may be malformed - e.g. if the user types something like:

   one
   l (Lower-case l, not the digit 1)
   O (Upper-case O, not the digit 0)

   For now, we will not talk about how to handle this - that comes later.

C. Thus, the problem of outputing and inputting numbers actually has two components - converting between internal (binary) and string representations of a number, and outputting or inputting the character string.

1. If format is not an issue, there are some easy mechanisms for converting an internal binary number to a human-readable string in a reasonable way.

   a) One method, provided by the objectdraw package, is to make use of the fact that Text objects can be created to display numbers as well as character Strings.

      Example: ICanCount

      DEMO

      SHOW CODE

      Modify to set font size to 24 (add setFontSize(24) to begin())

   b) Another method is to use the standard library variable System.out, which has methods print() and println()

      (1) print() simply writes the value it is asked to print. If several prints are done in a row, the output from one begins where the previous output leaves off.

      (2) println() outputs a newline after printing the value

         Example: I CanCountConsoleApp

         DEMO - run using main

         SHOW CODE

2. When one just wants to label numeric output. it is possible to use the operator +, which stands for addition when used with numbers, to convert numbers to character strings and concatenate them.

   Example:

Modify ICanCount to label the count with the string "The count is " - note space.

DEMO

(This can also be used with System.out.print(ln), but we won't demonstrate)

3. On the input side, the book talks about how to do this quite a bit later in the text. However, we will look at one way to do this now, using a standard Java classes called JOptionPane and Integer. (For now, treat this as boilerplate code)

   Example: JOptionPaneDemo

   DEMO

   SHOW CODE

   .

   DEMO what happens if a malformed number (e.g. "one") is typed - we will discuss dealing with issues like this later in the course.

VI. Random Numbers

A. The book discussed some mechanisms in the objectdraw library for creating "random numbers"

   In fact, these are not truly random - they really on an algorithmic technique known as a pseudorandom number generator.

   However, the results are indistinguishable from truly random results.

B. We will look at a simple example to illustrate the mechanism provided by object draw: A class written by the books' authors (though not included in their book) called Scribble

   DEMO

   SHOW CODE