CS112 Lecture: Variables, Expressions, Computation, Constants, Numeric
Input-Output and Mathematical Functions

Objectives:
1. To introduce variables and declarations of variables
2. To introduce assignment to variables
3. To explain the fundamental distinction between primitive types and reference types, and to introduce the Java primitive types
4. To introduce arithmetic operators, and expressions
5. To introduce literal and symbolic constants
6. To introduce mechanisms for reading and writing numbers using standard IO and JOptionPane, including formatting of output and parsing of input
7. To introduce the constants and functions of java.lang.Math

Materials:
1. BlueJ Bank Example demonstration
2. Transparency - Wu Figure 3.3
3. Transparency - Wu Table 3.1 (use 2nd edition version)
4. Transparency - Wu Table 3.2 (use 2nd edition version)
5. Transparency - Wu Table 3.3 (use 2nd edition version)
6. Dr. Java for use in demonstrations
7. Transparency - Wu Table 3.5
8. Demo programs: JOptionPaneDemo, StandardIODemo, NumberFormatDemo
9. Transparency - Wu Table 3.6 (use 2nd edition 3.5)

I. 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*.

(Note: Wu speaks of a variable having three properties, which correspond to the three characteristics that do not change, and are fixed when the variable is declared)

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

   ```java
   int i;
   float a, b, c;
   ```

3. A variable declaration may optionally include initialization for the value of the variable - e.g.

   ```java
   int i = 42;
   ```

   If a variable is not initialized when it is declared, in some contexts its value is undefined until it is given a value. The compiler will detect an attempt to use an uninitialized variable and will report an error if this is done. (In some contexts, uninitialized variables are given a default value - e.g. 0 for an integer.)

4. Your text stated that once a given name has been declared as the name of a variable, it cannot be used for a different variable. That is only part of a more complicated truth. The same name can be used for different variables in different *scopes* within the same program - i.e. two different methods may declare variables having the same name. (More on this later).

5. 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:

   ```java
   <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 absurd 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.
II. Data Types

A. Java is what we call a *strongly-typed* language - by which we mean that every variable is declared to hold only values of a particular data type, and may not be used to hold other types of values.

1. This stands in contrast with *untyped* languages (such as JavaScript), which allow the same variable to hold values of different types at different times during the execution of a program.

2. Strong-typing has two advantages:

   a) Efficiency. Untyped languages require that the actual type of a variable be checked every time the variable is used, because the meaning of an operation may depend on the type of operand it is applied to.

   b) Safety. Strong typing promotes more robust programs, because the compiler can catch certain errors that would otherwise cause the program to fail at runtime.

B. What are the permissible types for a variable in Java? Basically, they fall into two broad categories:

1. *Primitive*, or *value* types - the storage allocated for the variable holds its actual value.

2. *Reference* types - the storage allocated for the variable holds a reference to another region of memory where the entity the variable refers to actually resides.

3. We have been using reference type variables since the beginning of the course. When we declare an object - as in

   ```java
   Robot karel;
   ```

   or

   ```java
   JFrame myWindow;
   ```

   What we are doing is declaring a reference type variable which will be used to refer to an object of the appropriate class when that object is created. (That is, what we have been calling an object is actually a variable that refers to an object - but calling it an object is a convenient abbreviation)
4. Illustration of the difference

BlueJ Bank Example demo - create a Customer and a BankAccount, then inspect BankAccount; double click reference to bring up Customer object.

*TRANSPARENCY* - Wu. figure 3.3.

5. An important issue that arises with reference types but not with primitive types is *aliasing* - the possibility that two different variables might refer to the same object

Note how this occurs in final panel of Transparency

C. Although the number of different reference types that occur in a program is practically unlimited, since each Class constitutes a distinct type, the set of primitive types is fixed and defined by the language, and the names of these types are *reserved words* - words that can be used for no other purpose.

1. In this chapter, you were introduced to six primitive types that can be used to represent numbers with varying degrees of precision. What are they?

   a) *ASK*

   b) *TRANSPARENCY* - Wu table 3.1

2. You have actually already met another primitive type in our discussion of Karel.

   a) *ASK*

   b) The type boolean - whose possible values are false and true.

3. There is one more primitive type which we will meet later - the type char, whose possible values are individual characters - e.g. ‘A’ or ‘.’

III. 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.

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 Wu ch. 3 (there are others).

TRANSPARENCY - Wu. Table 3.2

1. Note that there are two kinds of operators listed:

   a) *Unary* operators take a single operand (as in -3)

   b) *Binary* operators require two operands (as in 2 + 2)

2. Note also that some operators are *overloaded* - they have more than one possible meaning, based on the context in which they are used.

   a) Example: + and - have both unary and binary arithmetic meanings

   b) Actually, + has an additional, non arithmetic meaning as well, which we will meet later.

   c) Technically, all the arithmetic operators are overloaded because they can be used for either integer arithmetic (both operands are integers, result is an integer) or real number arithmetic (at least one operand is a float or a double, result is a real).

   Example:

   3 / 2 is 1. (Integer division - discard remainder)

   3.0 / 2.0 is 1.5 (Real number division)

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

TRANSPARENCY - Wu Table 3.3

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)

\[- a + b \% c / d - e * - f - g\]

\[1 \quad 6 \quad 3 \quad 4 \quad 7 \quad 5 \quad 2 \quad 8\]

equivalent to

\[( ( ( - a ) + ((b \% c) / d)) ) - (e * (- f)) ) - g\]

4. 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

   NOTE: The book calls parentheses an operator with higher precedence than any other. Another way to view this - and one that I think is clearer - is to think of parentheses as making a whole expression (the one enclosed by the parentheses) be treated as a single operand.

D. Possible kinds of operands in Java expressions include:

1. Variables
2. Constants
3. Calls to methods that return a value of a type that is appropriate to appear in the expression at that point.

4. A parenthesized expression

Example: 2 \* (1 + Math.sin(theta))

includes all of the above

E. Just as variables have a type, so expressions have a type. The type of an expression is determined by the types of its operands and operators.

1. For the arithmetic operators we have learned thus far, the type of the result of operation is determined by the types of its operands according to a rule that is sometimes called the rule of numeric promotion.

   a) For unary operators

      (1) Operands of type byte or short are widened to int, and the result is of type int.

      (2) For all other types of operands, the type of the result is the same as the type of the operand.

   b) For binary operators:

      (1) If either operand is of type double, the other is converted to double (if necessary) and the result is of type double.

      (2) Otherwise, if either operand is of type float, the other is converted to float (if necessary) and the result is of type float.

      (3) Otherwise, if either operand is of type long, the other is converted to long (if necessary) and the result is of type long.

      (4) Otherwise, both operands are converted to int (if necessary) and the result is of type int.

   c) Note that the intent of the rule is to preserve the magnitude of the result. Also, types short and byte are available for conserving storage, but are not directly used in doing arithmetic (this relates to the architecture of the Java Virtual Machine, which does not support performing arithmetic operations directly on these types.)
d) Note that the rule is applied step by step during the evaluation of the expression.

Example: What is the value of the following expression

\[(4 / 5) * 2.0\]

\textit{ASK}

Answer: 0.0 !

The division \(4 / 5\) is done in type int, because both operands are ints. The quotient is 0 and the remainder of 4 is discarded. The int 0 is then converted to double 0.0 and multiplied to still yield 0!

2. A related requirement is that the type of an expression that is assigned to a variable be \textit{assignment compatible} with the variable.

a) A variable of integer type (byte, short, int, or long) may only be assigned an expression whose value is of the same type or a narrower type - e.g. byte can only accept byte; short can only accept byte or short; int can only accept byte, short, or int; and long can accept any integer type.

b) A variable of type float may be assigned an expression whose value is of type float, or of any integer type.

c) A variable of type double may be assigned an expression whose value is of any numeric type.

d) Again, the intent of the rule is to preserve the magnitude of the result.

F. There are times when it is necessary to perform an operation where the needed type conversion is not automatically performed. In this case, one must use an explicit \textit{type cast}.

Example: The following statement is erroneous, because the expression is of type int and the variable is of type byte:

\begin{verbatim}
byte i;
int j;
i = j + 1;  // Error
\end{verbatim}

To make this work correctly, we need to use a type cast, as follows:
i = (byte) (j + 1);

Note: An explicit cast is required, because this might result in the loss of information:

Example: Demo above with j = 255. Ask class what they think the value of i will be, then show.

The explicit use of a cast means “I know this could result in the loss of information, but in this case I know this won’t happen, or I’m prepared to accept it.”

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. long: same as int, but followed by an l or L.

4. double: a whole number part, a decimal point, a fractional part, and an exponent (e or E followed by a signed integer). Either the whole number part or the fractional part (but not both) may be omitted. The constant may optionally be followed by a d or D (to denote double).

Examples:
3.14159
3. (better written as 3.0)
.3 (better written as 0.3)
6.02E23
1.0D

5. float: same as double, but must be followed by an f or F
6. character strings (a type we have not formally met yet, but which we have needed to use from time to time) - a sequence of characters enclosed in double quotes (". If a quote is needed within the constant, it must be preceded by a backslash (\); it a backslash is needed, if must be doubled.

D. Symbolic constants are explicitly declared in the program

1. They are declared the same way variables are, except that
   a) The declaration includes the word final.
   b) 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:

   ```java
   final int MINIMUM_AGE = 18;
   final double DEGREES_PER_RADIAN = 180 / Math.PI;
   ```

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.

3. Why should we use symbolic constants?
   a) Readability - a constant with a good name is more understandable than a “magic number” - especially if 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. A number of the standard library classes define constants, as we shall see throughout the course. The chapter we read for today talked about two that are defined as a part of class java.lang.Math:

ASK
1. Math.E - the base of natural logarithms
2. Math.PI
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 1 (Lower-case l, not the digit 1)
O (Upper-case O, not the digit 0)

etc

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

C. Thus, the problem of outputting 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, Java provides some easy mechanisms for converting an internal binary number to a human-readable string in a reasonable way. In particular, the operator +, which stands for addition when used with numbers, can be used to convert numbers to character strings and concatenate them.

Examples: (DEMO with Dr. Java)
"You typed: " + 42 is equivalent to "You typed: 42"
"" + 42 is equivalent to "42"
"" + 42 + 3 is equivalent to "423"
"" + (42 + 3) is equivalent to "45"

(note how operator precedence affects the last two)

[ We'll talk about controlling format explicitly shortly ]

2. On the input side, Java provides “wrapper” classes to convert a character string to a binary number.

TRANSPARENCY: Wu Table 3.5
Demonstrate each case with Dr. Java (no semicolon after each line)
Note how erroneous cases throw an exception - we’ll discuss handling these later

D. The book discusses two general approaches to the inputting and outputting of character strings.

1. One approach makes use of the class JOptionPane in the Swing package.
   This class has a number of static “convenience” methods that create and show a GUI dialog box, and destroy it after the user dismisses it.

   a) The class JOptionPane includes the following method:
   
   ```java
   void showMessageDialog(parent, message)
   ```
   this creates a dialog to display a message, centered in the area on the screen occupied by the parent (which can be null). The dialog remains displayed until the user clicks OK.

   b) The class JOptionPane also includes the following method:
   
   ```java
   String showInputDialog(parent, message)
   ```
   this creates a dialog that displays the message and allows the user to type a response, which is returned to the caller by the dialog. The result is a character string, which must then be parsed as a number.

   c) Example: a program that shows the use of both methods:
   ```java
   JOptionPaneDemo.java
   ```
   (1) DEMO
   (2) Show code
2. The other approach makes use of two objects that are part of the standard Java library called System.in and System.out that can be used to do textual input-output to standard input and standard output.

   a) System.out has methods called print and println that can be used to print almost anything on standard output

      (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

      (3) Variants of print() and println() are defined for each of the primitive types that handle the conversion to strings for you.

   b) System.in can be used with two “wrapper classes” to read a string of characters from standard input

   c) Example: a program that demonstrates the above:
       StandardIODemo.java

       (1) DEMO

       (2) Show Code

E. Formatted Output

For producing formatted output of numbers, one can use the class java.text.DecimalFormat.

1. This class has a constructor which takes a parameter that specifies a format. For now we’ll look just at a subset of its capabilities.

2. DEMO: NumberIODemo

3. The format can include the following characters:

   # - print a digit - but suppress unnecessary 0’s
   0 - print a digit - but print a 0 if necessary
   . - decimal point
   , - grouping
VI. Mathematical Functions

A. One facility any general purpose programming language must provide is a mechanism for using standard mathematical functions.

B. In Java, for consistency with the style of the language, the standard mathematical functions are made available as class methods of the library class java.lang.Math. The package java.lang is special in the sense that classes in it do not need to be imported; they are automatically available to every Java program.

C. TRANSPARENCY - Wu Table 3.6

NOTE: The table is incomplete. Several methods are defined for double as well as for int, long, and float; there are a few minor methods not included.

D. Example: Given an angle theta expressed in degrees, calculate its sine (using constant DEGREES_PER_RADIAN defined above)

Math.sin(theta / DEGREES_PER_RADIAN);

VII. The book covers the class GregorianCalendar and the internal representation for various kinds of numbers; we will not discuss in class.