Variables, Values, and Expressions

The Class String

Reading:

=> Section 1.2
Variables are memory locations that store data such as numbers and letters.

The data stored by a variable is called its value.

A variables’ value can be changed.

A variable must be declared before it is used.
public class Cows
{
   public static void main (String[] args)
   {
      int barns, cowsPer, totalCows;

      barns = 10;
      cowsPer = 6;

      totalCows = barns * cowsPer;

      System.out.println("If you have");
      System.out.println(cowsPer);
      System.out.println(" cows per barn and");
      System.out.println(barns);
      System.out.println(" barns, then");
      System.out.println("the total number of cows is ");
      System.out.println(totalCows);
   }
}

Output:
If you have
6 cows per barn and
10 barns, then
the total number of cows is 60
public class Cows {

    public static void main (String[] args) {
        int barns, cowsPer, totalCows;

        barns = 10;
        cowsPer = 6;

        totalCows = barns * cowsPer;

        System.out.println ("If you have");
        System.out.println (cowsPer + " cows per barn and");
        System.out.println (barns + " barns, then");
        System.out.println ("the total number of cows is " + totalCows);
    }
}

**Output:**
If you have
6 cows per barn and
10 barns, then
the total number of cows is 60
Variables and Values

- Variables:
  
  barns
cowsPer
totalCows

- Assigning values: (must be declared before used)

  barns = 10;
cowsPer = 6;
totalCows = barns * cowsPer;

- When you *declare* a variable, you provide its name and *type*:

  ```
  int barns, cowsPer, totalCows;
  ```

- Can be declared separately:

  ```
  int barns;
  int cowsPer;
  int totalCows;
  ```
A variable’s type determines what kinds of values it can hold, e.g., integers, real numbers, characters.

Examples:

```java
int x, y, i, j;
double balance, interestRate;
char jointOrIndividual;
```

A variable is typically declared at the beginning of main:

```java
public static void main(String[] args)
{
    -- declare variables here --
    :
}
```
An **identifier** is a name given to something in a program:

- a variable, method, class, etc.
- created by the programmer, generally

Identifiers may contain only:

- letters
- digits (0 through 9)
- the underscore character (_)
- and the dollar sign symbol ($) which has a special meaning (do not use!)
- *the first character cannot be a digit.*
Java Identifiers

Legal or not?

- count
- Name
- birch
- 1count
- x
- count1
- 7-11
- Cou2nt
- netscape.com
- myFavoriteHobby
- My_Favorite_Hobby
- _value1
- util.*
- i

(birch trees)
Identifiers can be arbitrarily long.

```java
int myFavoriteIdentifierIsCompletelyUseless;
```

Java is *case sensitive* i.e., `stuff`, `Stuff`, and `STUFF` are different identifiers.

*Keywords or reserved words* have special, predefined meanings:

abstract assert boolean break byte case catch char class const continue
default do double else enum extends false final finally float for goto if
implements import instanceof int interface long native new null package
private protected public return short static strictfp super switch
synchronized this throw throws transient true try void volatile while

Keywords *cannot* be used as identifiers (but they can be used as *part of* an identifier; use with caution).
It is important to distinguish between the rules of the language and naming conventions.

- **Rules of the language** – enforced by the compiler.

- **Naming conventions** – commonly used guidelines that are typically a matter of style, but not enforced by the compiler.
A common recommendation is to choose names that are readable, such as `count` or `speed`, but not `c` or `s`.

That having been said, sometimes short names are ok; `x`, `y`, `i`, `j`, `k`.

Variables:
- begin with a lowercase letters (e.g. `myName`, `myBalance`).
- multiword names are “punctuated” using uppercase letters.
In most programming languages, a *type* implies several things:

- A set of values
- A (hardware) representation for those values
- A set of operations on those values

**Example - type** *int*:

- Values – integers in the range -2147483648 to 2147483647
- Representation – 4 bytes, binary, “two’s complement”
- Operations – addition (+), subtraction (-), multiplication (*), division (/), and remainder (%).

**Two kinds of types:**

- Primitive types
- Class types
Types in Java

- **A primitive type:**
  - values are “simple,” non-decomposable values such as an individual number or character
  - int, double, char

- **A class type:**
  - values are “complex” objects
  - ‘November 10, 1989’ is a value of class type date (non-Java)
  - ‘150 West University Blvd., Melbourne, FL, 32901’ is a value of class Address
  - “BIG bad John.” is a value of class type String
  - a class has “methods,” i.e., operations
This semester, we will make use of four primitive types:
- int
- double
- char
- boolean

And two class types:
- String
- Scanner

CS1002 will cover many more types, including class types.
Examples of Primitive Values

- Integer values:
  0  -1  365  12000

- Floating-point values:
  0.99  -22.8  3.14159  5.0  4.83e-2  0.25e5

- Character values: (enclosed in single quotes)
  `a`  `A`  `#`  ` `

- Boolean values:
  true  false

- Values such as the above are often referred to as constants or literals.
As noted previously, an assignment statement is used to assign a value to a variable:

```java
int answer;
answer = 42;
```

The “equal sign” is called the *assignment operator*. 
Assignment syntax:

```
variable = expression;
```

where *expression* can be

- a *literal* or *constant* (such as a number),
- another variable, or
- an expression which combines variables and literals using operators
Examples:

```c
int amount;
int score, numOfCards, handicap;
int eggsPer;
char firstInitial;
double pie;

score = 3;
umOfCards = 25;
handicap = 5;
firstInitial = 'W';
eggsPer = 6;
pie = 3.14;
amount = score;
score = numOfCards + handicap;
eggsPer = eggsPer - 2;
```

⇒ Some think the last line looks weird in mathematics. Why?
At a high-level, an assignment consists of two steps:
- evaluation of the RHS
- assignment to the LHS

The expression on the right-hand side of the assignment operator (=) is evaluated first.

The result is used to set the value of the variable on the left-hand side of the assignment operator.

```plaintext
score = numberOfCards + handicap;
eggsPerBasket = eggsPerBasket - 2;
```

The distinction between the two steps is very important!
Java is sometimes said to be *strongly typed*, which means that there are limitations on mixing variables and values in expressions and assignments.

```java
int x;
double y;
String name;

x = 10;
y = 37.5;
name = "bob";
x = 5.3;  // Error: incompatible types
x = y;    // Error: incompatible types
x = 20;
y = x;
y = 25;
x = name;  // Error: incompatible types
y = name;  // Error: incompatible types
name = y;  // Error: incompatible types
name = x;  // Error: incompatible types
```
Java is said to be *strongly typed*, which means that there are limitations on mixing variables and values in expressions and assignments.

```java
int x;
double y;
String name;

x = 10;
y = 37.5;
name = "bob";
x = 5.3; // Can’t assign a double to an int
x = y;   // Can’t assign an int to a double
x = 20;
y = x;
y = 25;
x = name; // Can’t assign a string to an int
y = name; // Can’t assign a string to a double
name = y; // Can’t assign a double to a string
name = x; // Can’t assign an int to a string
```
There are some interesting exceptions to the typing rules…

Recall that, like everything else, each character is represented by a binary sequence.

The binary sequence corresponding to a character is a positive integer.

Which integer represents each character is dictated by a *standardized* encoding.
- Each character is assigned a unique integer code
- The codes are different for upper and lower case letters, e.g., 97 is the integer value for ‘a’ and 65 for ‘A’

Why should different computers and languages use the same code?
Most programming languages use the ASCII character encoding.

- American Standard Code for Information Interchange (ASCII)
- (only) encodes characters from the north American keyboard
- uses one byte of storage

*Hey, let's google ASCII!!!!*

Java uses the *Unicode* character encoding.

- The Unicode character set:
  - uses two bytes of storage
  - includes many international character sets (in contrast to ASCII)
  - codes characters from the North American keyboard the same way that ASCII does
Assigning a `char` to an `int`

- Getting back to the “type” issue...a value of type `char` can be assigned to a variable of type `int` to obtain its Unicode value.

- Example:

```java
char ch;
int x;

ch = 'a';
System.out.println(ch);  // Outputs ‘a’
x = ch;
System.out.println(x);  // Outputs 97
```

- While it might seem a bit odd, this can be very helpful for character processing.
A variable that has been declared, but not yet given a value is said to be *uninitialized*.

```c
int x, y, z;
x = y;
x = z + 1;
```

Some languages automatically initialize a variable when it’s declared, other languages don’t.
Some languages give a compile time error if a variable is not initialized.
- Program does not compile or run.

Some languages give a compile time warning if a variable is not initialized.
- Program compiles and runs.
- The initial value of the variable is typically arbitrary!
- The program might appear to run correctly sometimes, but give errors on others.

Some languages don’t give either, and automatically initialize.

Some languages don’t give either, and don’t automatically initialize:
- The initial value of the variable is arbitrary!
- The program might appear to run correctly sometimes, but give errors on others.
Initializing Variables

- **Java:**
  - The compiler will (try) to identify uninitialized variables.
  - In some cases, Java will automatically initialize an uninitialized variable.
  - In other cases, the compiler will complain that a variable is uninitialized, when in fact, is isn’t; in such a case, initializing it in its’ declaration usually solves the problem.

- **Examples:**

```java
int count = 0;
char grade = 'A';
```

=> *Always make sure your variables are initialized prior to use!*
Arithmetic expressions:
- Formed using the +, -, *, / and % operators
- Operators have operands, which are literals, variables or sub-expressions.

Expressions with two or more operators can be viewed as a series of steps, each involving only two operands.
- The result of one step produces an operand which is used in the next step.
- Most of the basic rules of precedence apply.
- Java is left-associative.
Arithmetic Operations

Example:

```java
int x = 0, y = 50, z = 20;
double balance = 50.25, rate = 0.05;

x = x + y + z;
balance = balance + balance * rate;
balance = (balance + balance) * rate;
```

Some vocabulary:
- operator
- operand
- expression
A single expression can contain operands of both double and int types.

In that case, the type of the resulting expressing is double.

```java
int hoursWorked  = 40;
double payRate   = 8.25;
double totalPay;
```

Then the expression in the assignment:

```java
totalPay = hoursWorked * payRate
```

results in a double with a value of 330.0.

If the variable `totalPay` is of type `int`, then the above statement does not compile.
The Division Operator

- The division operator (/) behaves as expected.

- If at least one of the operands is a double then the result is type double:
  - \( 9.0 / 2 = 4.5 \)
  - \( 9 / 2.0 = 4.5 \)
  - \( 9.0 / 2.0 = 4.5 \)
  - \( 10.0 / 2 = 5.0 \)

- If both operands are type int then the result is truncated (not rounded) and of type int.
  - \( 9 / 2 = 4 \)
  - \( 99 / 100 = 0 \)
This is true, regardless of whether the operands are literals, variables, or more general expressions.

```c
int x = 40;
int y = 7;
double z = 8.25;
double w = 2.5;
```

If at least one of the operands is a double then the result is type double:

- \(z / x = 0.20625\)
- \(y / 2.5 = 2.8\)
- \(y / w = 2.8\)
- \((y + 1) / w = 3.2\)
- \(z / w = 3.3\)

If both operands are type int then the result is **truncated** and of type int.

- \(x / y = 5\)
- \(x / 7 = 5\)
- \((x + 1) / 8 = 5\)
The mod (%) operator is used with operands of integer type to obtain the remainder after integer division.

14 divided by 4 is 3 with a remainder of 2.
- Hence, $14 \mod 4$ is equal to 2.

The mod operator has many uses, including determining:
- If an integer is odd or even ($x \mod 2 = 0$)
- If one integer is evenly divisible by another integer ($a \mod b = 0$)
- Frequently we want to map (a large number of) $m$ “items,” number 1 through $m$, into $n \geq 2$ groups or “buckets.”
Program Requirements:
- The user enters an amount between 1 cent and 99 cents.
- The program determines a combination of coins equal to that amount.
- For example, 55 cents can be two quarters and one nickel.
- Largest denominations are given preference.

Sample dialog:
Enter an integer from 1 to 99: 87
87 cents in coins is:
  3 quarters
  1 dime
  0 nickels
  2 pennies

=> What are “requirements” anyway?
How do we determine the number of quarters in an amount?

- Use integer division:
  
  \[
  \begin{align*}
  55 \div 25 &= 2 \\
  95 \div 25 &= 3 \\
  40 \div 25 &= 1 \\
  65 \div 25 &= 2
  \end{align*}
  \]

How do we determine the remaining amount?

- Use the mod operator:
  
  \[
  \begin{align*}
  55 \% 25 &= 5 \\
  95 \% 25 &= 20 \\
  40 \% 25 &= 15 \\
  65 \% 25 &= 15
  \end{align*}
  \]

Similarly for dimes, nickels and pennies
import java.util.Scanner;
public class VendingMachine {
    public static void main(String[] args) {
        int amount, originalAmount, quarters, dimes, nickels, pennies;
        Scanner kb = new Scanner(System.in);

        System.out.print("Enter an integer between 1 and 99:");
        amount = kb.nextInt();
        originalAmount = amount; // Save the original amount for later

        quarters = amount/25;
        amount  = amount%25;
        dimes   = amount/10;
        amount  = amount%10;
        nickels = amount/5;
        pennies = amount%5;

        System.out.println(originalAmount + " cents in coins is:");
        System.out.println(quarters + "quarters");
        System.out.println(dimes + "dimes");
        System.out.println(nickels + "nickles");
        System.out.println(pennies + "pennies");
    }
}
As we already have seen, a String is a sequence of characters.

We’ve used constants, or rather, literals of type String:

“Enter a whole number from 1 to 99.”
“Number of quarters:”
“I will output a combination of coins”
Declaring and Printing Strings

- Variables of type String can be declared and Initialized:

  ```java
  String greeting;
  greeting = “Hello!”;
  ```

- Equivalent to the above:

  ```java
  String greeting = “Hello!”;
  String greeting = new String (“Hello!”);
  ```
Variables of type String can be input & output:

```java
String Word;
System.out.print("Enter a word: ");
Word = keyboard.next();
System.out.print("The word entered was: ");
System.out.println(Word);
```

> Enter a word: dog
> The word entered was: dog
concatenation of strings

- Two strings can be *concatenated* using the + operator:

  ```java
  String s1 = "Hello";
  String s2 = "Rogers";
  String s3;

  s3 = s1 + " officer ";
  s3 = s3 + s2;
  ```

- Any number of strings can be concatenated using the + operator.
Strings can be concatenated with other types:

```java
String solution;

solution = "The temp is " + 72;
System.out.println (solution);
```

Output:

```
The temp is 72
```
Recall that Java has primitive types and class types:

- **primitive types**
- **class types**

Primitive types have **operations** (built-in, usually a symbol such as *, +, or -)

Class types have **methods** (some built-in, others user-defined, usually a word)
String is a class type, which means it has methods.

The `length()` method returns an `int`, which is the number of characters in a particular `String` object.

```java
String solution = "dog";
int count;
count = solution.length();
System.out.println(count);
```

You can use a call to method `length()` anywhere an `int` can be used.

```java
int x = solution.length();
x = x * solution.length() + 3;
System.out.println(solution.length());
```
Positions in a String

- Each character in a String has its own *position*.

- Positions are numbered starting at 0.
  - ‘J’ in “Java is fun.” is in position 0
  - ‘f’ in “Java is fun.” is in position 8

- The position of a character is also referred to as its *index*. 
The twelve characters in the string "Java is fun." have indices 0 through 11. The index of each character is shown above it.

```
0  1  2  3  4  5  6  7  8  9  10  11
J a v a   i s   f u n  .
```

Note that the blanks and the period count as characters in the string.

Display 2.8
String Indices
String Method #2 – charAt

- `charAt(position)`
  - returns the char at the specified position

- Example:

```java
String greeting = "Hi, there!";  
char ch1, ch2, ch3;  

ch1 = greeting.charAt(0);  // Stores ‘H’ in ch1  
ch2 = greeting.charAt(2);  // Stores ‘,’ in ch2  
ch3 = greeting.charAt(10);  // Oops!
```
String Method #3 – substring

- substring(start, end)
  - returns the string from start up to, but not including, end

Example:

String myWord;
String greeting = "Hi, there!";

myWord = greeting.substring(4,7);  // Stores the in myWord
System.out.println(myWord);

> the
String Method #4 – \texttt{indexOf}

- **\texttt{indexOf(str)}**
  - returns the starting position of string \texttt{str}

**Example:**

```java
int pos;
String phrase = "The cow is an old cow";

pos = phrase.indexOf("an");
System.out.print(pos);
pos = phrase.indexOf("cow");
System.out.print(pos);
pos = phrase.indexOf("old");
System.out.print(pos);
pos = phrase.indexOf("dog");
System.out.println(pos);
```

> 11 4 14 -1
<table>
<thead>
<tr>
<th>toLowerCase()</th>
</tr>
</thead>
<tbody>
<tr>
<td>returns a copy of the string converted to lowercase</td>
</tr>
</tbody>
</table>

**Example:**

```java
String phrase = "SaMpLE StriNg";
String lcPhrase;

System.out.println("Before: " + phrase);
lcPhrase = phrase.toLowerCase();
System.out.println("After: " + lcPhrase);

>Before: SaMpLE StriNg
>After: sample string
```
There are many Java String methods.

See the Java on-line documentation for more details!
Escape Characters

How would you print the following?

"Java" refers to a language.

The following don’t work:

```java
System.out.println("Java refers to a language.");
System.out.println(""Java" refers to a language.");
```

The compiler needs to be told that the quotation marks (") do not signal the start or end of a string, but instead are to be printed.

```java
System.out.println("\"Java\" refers to a language.");
```
“Escape sequences” are used to print “problematic” characters.

\"        Double quote
\'        Single quote
\\       Backslash
\n        Newline (beginning of next line)
\r        Carriage return (beginning of current line)
\t        tab
\f        form-feed (beginning of next page)
\b        backspace
Examples:

```
System.out.println("dognhair"); => dognhair

System.out.println("dog\nhair"); => dog hair

System.out.println("dog\\nhair"); => dog\nhair

System.out.println("dog\tha\r"); => dog hair

System.out.println("dog\dhair"); => ???

char singleQuote = '\";
System.out.println(singleQuote); => \\
```
What do the following do?

```java
String s1, s2;

s1 = "The\ndog";
System.out.println(s1.length());
System.out.println(s1);

s2 = "The\ndcar";
System.out.println(s2);
```
public class StringDemo {
    public static void main(String[] args) {
        int position;
        String sentence = "Text processing is difficult!";
        System.out.println(sentence);

        // Find the position of the word "difficult"
        position = sentence.indexOf("difficult");
        System.out.println("The word "difficult" starts at location " + position);

        // Replace the word "difficult" with "easy" and output
        sentence = sentence.substring(0, position) + "easy!";
        System.out.println("The changed string is:");
        System.out.println(sentence);

        // Find the position of the word "is"
        position = sentence.indexOf("is");
        System.out.println("The word "is" starts at location " + position);

        // Replace the phrase "Text processing" with the word "College" and output
        sentence = "College " + sentence.substring(position, sentence.length());
        System.out.println("The changed string is:");
        System.out.println(sentence);
    }
}
Modify the previous program so that it prompts the user for a sentence and two words. The program will modify the sentence variable by replacing the first occurrence of the first word with the second word.

Enter a sentence: Gosh sunny, it sure is sunny today.
Enter a word: sunny
Enter another word: bob
Resulting sentence: Gosh bob, it sure is sunny today.
Increment (and Decrement) Operators

- Used to increase (or decrease) the value of a variable by 1.

```java
count = count + 1;
count = count - 1;
```

- The increment and decrement operations are easy to use, and important to recognize.

- The increment operator:

```java
count++;                // Postfix/suffix notation
++count;               // Prefix notation
```

- The decrement operator:

```java
count--;                // Postfix/suffix notation
--count;               // Prefix notation
```
Increment (and Decrement) Operators

- “Mostly” equivalent operations:

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

- Unlike the assignment versions, however, the increment and decrement operators are *operators* and have a resulting *value*...huh?
Consider the following declaration:

```
int m, result;
```

After executing:

```
    m = 4;
    result = 3 * (++m);
```

result has a value of 15 and m has a value of 5

Similarly:

```
    result = 3 * ++m;
    result=3*++m;
```

// Note the role of precedence!
After executing:

\[
m = 4;
\]

\[
result = 3 \times (m++);
\]

result has a value of 12 and \( m \) has a value of 5
Increment and Decrement Operator, Cont.

- Assume the following declarations:
  ```
  int n = 3;
  int m = 4;
  int result;
  ```

- What will be the value of \( m \) and \( \text{result} \) after each of these executes?
  (a) \( \text{result} = n \ast \text{++m}; \)
  (b) \( \text{result} = n \ast \text{m++;} \)
  (c) \( \text{result} = n \ast \text{--m;} \)
  (d) \( \text{result} = n \ast \text{m--;} \)
  (e) \( \text{result} = \text{++m} \ast n; \)
  (f) \( \text{result} = \text{m++;} \ast n; \)
  (g) \( \text{result} = \text{--m;} \ast n; \)
  (h) \( \text{result} = \text{m--;} \ast n; \)
What about this:

```java
int x = 5;
x = x++;
System.out.println(x);
```

```java
int x = 5;
x = ++x;
System.out.println(x);
```