**Administrative**

**Today’s session**

printf method

Keyboard input

Addition/concatenation operator

Division operator

Variable

Constant

Assignment statement

Arithmetic expressions

Assignment compatibilities

Software errors

Homework 1

**Session Topics**

**printf method**

● The **printf** method formats output and then sends it to the screen.

● The printf method has syntax:

**System.out.printf(<string-expression-with-format-specifiers>, <expression-list>);**

Where …

**<string-expression-with-format-specifiers>** is some combination of character, string, and/or numeric expressions, and **format specifiers** connected by concatenation operators.

**<expression-list>** is a sequence of character, string, and/or numeric expressions separated by commas.

● There must be one format specifier for each expression in the <expression-list>.

● <format-specifier> controls one output column and:

✓ Specifies whether the value in the output column will be left- or right-justified.

✓ Specifies how wide to make an output column.

✓ If the value is a real number, specifies how many decimal places to round to and display in the output column.

✓ Specifies the data type of the expression to be printed.

✓ Has syntax:

**%<flag><number><converter>**

Where …

**<flag>** may be:

| Flag | Purpose |
| --- | --- |
| + | Include a number sign whether positive or negative. |
| , | Include commas in a number. |
| - | Left-justify the value within the column. |

**<number>** may be:

| Number | Purpose |
| --- | --- |
| N or 0N | Output the value in a column N characters wide. If N is omitted or is not large enough for the value, the column will be made big enough to fit the value. If 0 is included, leading zeroes will be shown. |
| N.M or 0N.M | (conversion code f only) Output the real number value in a column N characters wide rounded to M decimal places. If M is omitted, the value is rounded to six decimal places. If 0 is included, leading zeroes will be shown. |

**<converter>** is one of:

| Converter | Purpose |
| --- | --- |
| b | Format corresponding expression as boolean (true or false) |
| d | Format corresponding expression as an integer. |
| f | Format corresponding expression as a real number. |
| c | Format corresponding expression as a char. |
| s | Format corresponding expression as a string. |
| n | Move output cursor to start of new line. |

● The data type of each format specifier must match the data type of the corresponding expression.

● To include a percent sign (%) in the output, place two percent signs together (%%).

● If there is a block of printf statements using the same format specifiers:

1) Store each specifier in a string variable.

2) Use the variables, instead of the specifiers, in the printf statements.

● See **Formatted print** sample application on Blackboard.

**Keyboard input**

● The Scanner class is needed to read data from the keyboard.

● The Scanner class requires:

✓ An import statement at the top of the program:

**import java.util.Scanner;**

✓ A “keyboard” object declaration:

**Scanner keyboard = new Scanner(System.in);**

✓ Scanner methods to read data from the keyboard.

✓ A keyboard close when done with “keyboard” object:

**keyboard.close();**

● Scanner(System.in) object notes:

✓ There should only be one "keyboard" object declaration per application.

✓ There can only be one "keyboard.close" call per application.

● Scanner methods are used to store input provided by the user into a variable.

● A scanner method has syntax:

**<Scanner-variable>.<Scanner-method>();**

Where …

**<Scanner-variable>** is declared prior to the method call; is often called “keyboard”.

**<Scanner-method>** is a valid Scanner method.

● Many of these methods operate on chunks of data called **tokens** and **lines**.

● A **token** is zero or more characters starting and ending with (but not including) a whitespace character.

● A **whitespace character** could be:

| Character | Escape sequence | Unicode number |
| --- | --- | --- |
| Common whitespace characters | | |
| space |  | U+0020 |
| horizontal tab | \t | U+0009 |
| enter / return\* | \n | (depends on operating system) |
| Uncommon whitespace characters | | |
| line feed | \u000A | U+000A |
| vertical tab | \u000B | U+000B |
| form feed | \f | U+000C |
| carriage return | \r | U+000D |
| file separator\*\* |  | (depends on operating system) |
| group separator | \u001D | U+001D |
| record separator | \u001E | U+001E |
| unit separator | \u001F | U+001F |

\* *Windows:* line feed and carriage return; *Unix* and *Mac OS*: line feed

\*\* *Windows:* '\'; *Unix:* '/'; *Mac OS:* ':'

● A **delimiter** is a whitespace character used to determine where a token begins and ends. By default, this is the space character.

● A **line** is zero or more characters ending with (but not including):

✓ Keyboard input only: the ENTER key.

✓ File input only: the new-line (\n) character.

● These methods use **whitespace** to determine where a token begins and ends.

● Boolean **method java.lang.Character.isWhitespace()** may be used to determine whether a character read is a whitespace character or not.

● Here are common Scannermethods:

| Method | Description when used for keyboard input |
| --- | --- |
| nextInt()  nextDouble()  next() | Reads the next token from the buffer. If one is not available, it pauses the program to allow the user to enter one at the keyboard. It will scan past any whitespace characters until it reads a token. It then converts the token to an **int/double/string**. Any whitespace characters after the token, including the ENTER key, remain in the buffer. |
| nextLine() | Reads the next line from the buffer. If one is not available, it pauses the program to allow the user to enter one at the keyboard. It will NOT scan past any whitespace characters to begin reading the line. It then converts all characters read, except the ENTER key, to a string. It then discards the ENTER key from the buffer. |

● See **Keyboard input** sample application on Blackboard.

**Addition/concatenation operator**

● The plus operator (+) may mean addition or concatenation. Within an expression:

✓ If either side of the + operator is not numeric, then **concatenation** is performed.

✓ If both sides of the + operator are numeric, then **addition** is performed.

✓ Expressions and operators are processed left to right unless overridden by parentheses.

**Division operator**

● The division operator (/) will give an integer result if both operands are integer. This may be overridden by casting one of the operands to a floating-point type.

● See **Division operator** sample application on Blackboard.

**Variable**

● A **variable** is a named spot in memory used to hold a value.

● A variable has at least three characteristics:

✓ Name

→ Begins with a letter, dollar sign, or the underscore.

→ Is optionally followed by any sequence of letters, digits, dollar signs, or underscores.

→ May not be a keyword or reserved word. See Oracle’s list of [Java Language Keywords](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html).

✓ Value

✓ Data type (primitive data types plus String)

| Data Type | Range |
| --- | --- |
| Integer types | |
| byte | Integer from -128 to 127 |
| short | Integer from -32,768 to 32,767 |
| int | Integer from -2,147,483,648 to 2,147,483,647 |
| long | Integer from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| Real types | |
| float | Real number from 1.40e-45 to 3.40+38 (positive or negative) |
| double | Real number from 4.94-324 to 1.79+308 (positive or negative) |
| Other types | |
| boolean | **true** or **false** |
| char | [Unicode character](http://unicode-table.com/en/), from 0 to 65,535 (or hex FFFF) |
| String | A sequence of 0 to 2,147,483,647 characters |

● A **variable declaration** creates a variable in memory.

● A variable declaration has syntax:

**<data type> <variable-name>;**

OR

**<data type> <variable-name> = <value>;**

● See **Factorial calculator** sample application on Blackboard.

**Constant**

● A constant is identical to a variable except its initial value cannot be changed while a program is running.

● A constant has at least three characteristics:

✓ Name

✓ Value

✓ Data type (primitive data types plus String)

● A **constant declaration** creates a constant in memory.

● A constant declaration has syntax:

**final <data type> <constant-name> = <value>;**

**Assignment statement**

● An **assignment statement** places a value into a variable.

● An assignment statement has syntax:

**<variable> = <expression>;**

**Arithmetic expressions**

● Are made up of numbers, variables, and/or method calls, and arithmetic operators.

| Arithmetic Operator | Operation |
| --- | --- |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division (result is real unless both operands integer) |
| % | Modulo: remainder after division |

**Assignment compatibilities**

● When an assignment statement has different data types on each side of the equals sign, we must be aware what is stored, if anything, in the left-side variable.

| Variable data type | Expression data type | | | |
| --- | --- | --- | --- | --- |
|  | int | double | char | boolean |
| int | Stores int. | **Causes compilation error** | Stores Unicode value in int. | **Causes compilation error** |
| double | Stores int as double. | Stores double. | Stores Unicode value in double. | **Causes compilation error** |
| char | Stores Unicode character in char. | **Causes compilation error** | Stores char. | **Causes compilation error** |
| boolean | **Causes compilation error** | **Causes compilation error** | **Causes compilation error** | Stores boolean. |

**Software errors**

● There are three kinds of software errors:

✓ **Compilation** errors result when we violate the **syntax** of a programming language.

✓ **Run-time** errors result when our application attempts to divide by zero, exceed the bounds of an array, etc.

✓ **Logic** errors result when our application performs a calculation or handles data incorrectly, shows the wrong information on a screen, etc.

**Homework 1**

● Assigned today.

● Available on Blackboard.

● Due in one week.