User Input, Parameters, Methods that Return Values, Conditional Execution, Indefinite Loops, Commenting Style

CSCI 161 – Introduction to Programming I

Professor Thomas Rogers

Overview

- Reading: Chapter 3 - Introduction to Parameters and Objects, Chapter 4.1, Chapter 5.1

- Topics:
  - User Input
  - Parameters
  - Methods that Return Values
  - Conditional Execution
  - Indefinite Loops
  - Commenting Styles

User Input

- Input with the user is done through what is called **standard input**, which is a fancy way of saying the user typing in entries on the keyboard.
- User input is also called **Console Input** - responses typed by the user when an interactive program pauses for input.
- For the most part in this class we will be dealing with *"line at a time"* entry, meaning your programs will get and process a single line of input from the user at a time. Again, a single line of input is everything the user types up until the return key is pressed.

User Input (continued)

- The *Scanner* class is used for user input handling:

  - Before the Scanner class may be used, it must be made known to your Java program. This is done by using the **import** statement at the top of your program, like so:

```
import java.util.Scanner;
```

User Input (continued)

- To use the Scanner class create an object instance and pass in the type of scanner instance you desire:

```
Scanner console = new Scanner(System.in);
```

  - Where the object variable of the Scanner class instance is identified as **console**. (Could be any name).
  - The **new** keyword is used to indicate a new instance of an object is to be instantiated.
  - The parameter **System.in** is passed to the constructor of the object indicating the IO stream to use (standard input).

User Input (continued)

- The Scanner object has methods that control how it reads input:

Method	Description
next()	reads and returns the next token as a string
nextDouble()	reads and returns a double value
nextInt()	reads and returns an int value
nextLine	reads and returns the next line of input as a String

User Input (continued)

- **Token** - A single element of input (e.g. one word, one number) separated by **whitespace**.
- **Whitespace** - Spaces, tab characters, and new line characters.
- **WARNING:** If your program executes the nextInt() method of the Scanner object and the user types in something other than an integer you program will generate an exception (run-time error).
- **WARNING:** Your programs should create one Scanner object in the main method and pass it to methods that need to read input. Attempting to create the Scanner object within a method that is called successively will work when processing user input, but will not work when a file is redirected to standard input (which is how your labs are run when grading).
- **WARNING:** Your program should *not* declare its Scanner object within a loop, even a loop within the main method. Your program will work with user input from the Console but not with input from file redirection (how your program is graded) if your Scanner is declared in a loop.

User Input (continued)

- Redirecting a file to standard input is how your labs and assignments that require input will be run when graded. You should create your own data files for input and redirect them to your running program with the following syntax and test prior to submitting:

```
java Lab2 < myData.txt
```

  Where in the above, **myData.txt** is the text file containing your input data for testing.

- Always prompt the user with a meaningful prompt telling the user what type of input your program needs and any special values before using the scanner to get the input, as shown below:

```
System.out.print("\nEnter a number of seconds: ");
int seconds = console.nextInt();
```

Parameters

- **Formal parameter** - a variable that appears inside parentheses in the header of a method that is used to generalize the method's behavior (aka "parameter").
- **Actual parameter** - a specific value or expression that appears inside parentheses in a method call (aka "argument")
- A method can have zero, one or more parameters as so:

```
public static void foo()...

public static void bar( int num )...

public static void fubar( int num, String name )...
```

- When a method has multiple parameters, it is important to supply **arguments** to it in the correct order, with each having the correct data type when calling said method.

Parameters (continued)

- **Limitations of Parameters:** Arguments passed into a method can be used within the method and their values changed within the method, but any changes to said values are not reflected in the variable outside the method. In programming language terms this is what is known as parameters that are *passed by value*.
- **Method Overloading:** The ability to define two or more different methods with the same name but different method signatures, where the signature is the name of the method along with its specific combination of its number of parameters and their types.

Methods that Return Values

- One way to overcome the "pass by value" aspect of Java method parameters is to use method return values:

```
// Bad method
public static void upperCase( String str ) {
        str = str.toUpperCase();
}

String str = "hello";
upperCase( str );  // Nope - str remains "hello"

// Good method
public static String upperCase( String str ) {
        return str.toUpperCase();
}

String str = "hello";
str = upperCase( str ); // This works.
```

Methods that Return Values (continued)

- Void methods do not and cannot return any values.
- The return value is returned by your method through the use of the **return** statement.
- No code (or statement) below (after) the return statement executes within your method, even if within a conditional or loop.
- **Methods that return booleans:** It is often useful to define methods that return booleans to aid in the conditional execution and logic of your program. For example:

```
public static boolean isValidInput(String str)...
```

returns true if the supplied string is valid, else it returns false.

Conditional Execution

- One of the cornerstones of almost any *useful* program is its use of conditionals to drive different logical pathways of its execution. In other words, your programs will have some code that you want to run some of the time, under certain conditions or when encountering certain input and other times, with other conditions and other input you want some othe code to be executed or maybe no code at all to be executed.
- **if/else** The if/else statements provide the conditionals for just such execution.
- Sample if statement:

```
if (currentScore > maxScore) {
        System.out.println("A new high score!");
        maxScore = currentScore;
}
```

Conditional Execution (continued)

- Basic syntax is:

```
if (<test>) {
    <statement>;
    <statement>;
       ...
    <statement>;
}
```

- If the **test** logical expression evaluates to true then the controlled statements are executed and if not the program execution continues with statements after the controlled statements (after the closing curly brace).

Conditional Execution (continued)

- The **else** statement is optional but when used in association with the **if** provides a structure for defining the controlled statements to be executed when the test expression evaluates to false, like so:

```
if (<test>) {
   <statement>;
   <statement>;
      ...
   <statement>;
} else {
   <statement>;
   <statement>;
     ...
   <statement>;
}
```

- **IMPORTANT:** Always include the curly braces for **if** and **else** statements even if there is only one controlled statement.

Conditional Execution (continued)

- The if/else statement can use **Relational Operators** to define the logical expression that is evaluated:

```
<expression> <relational operator> <expression>
numChars < MAX_LENGTH
x >= 4
```

- **LOOK IN THE BOOK!** See tables 4.1 and 4.2 in book for list of relational operators, Java operator precedence and examples.
- **LOOK IN THE BOOK!** Read up on nested if/else statements, how they work, why to use them (especially table 4.3).

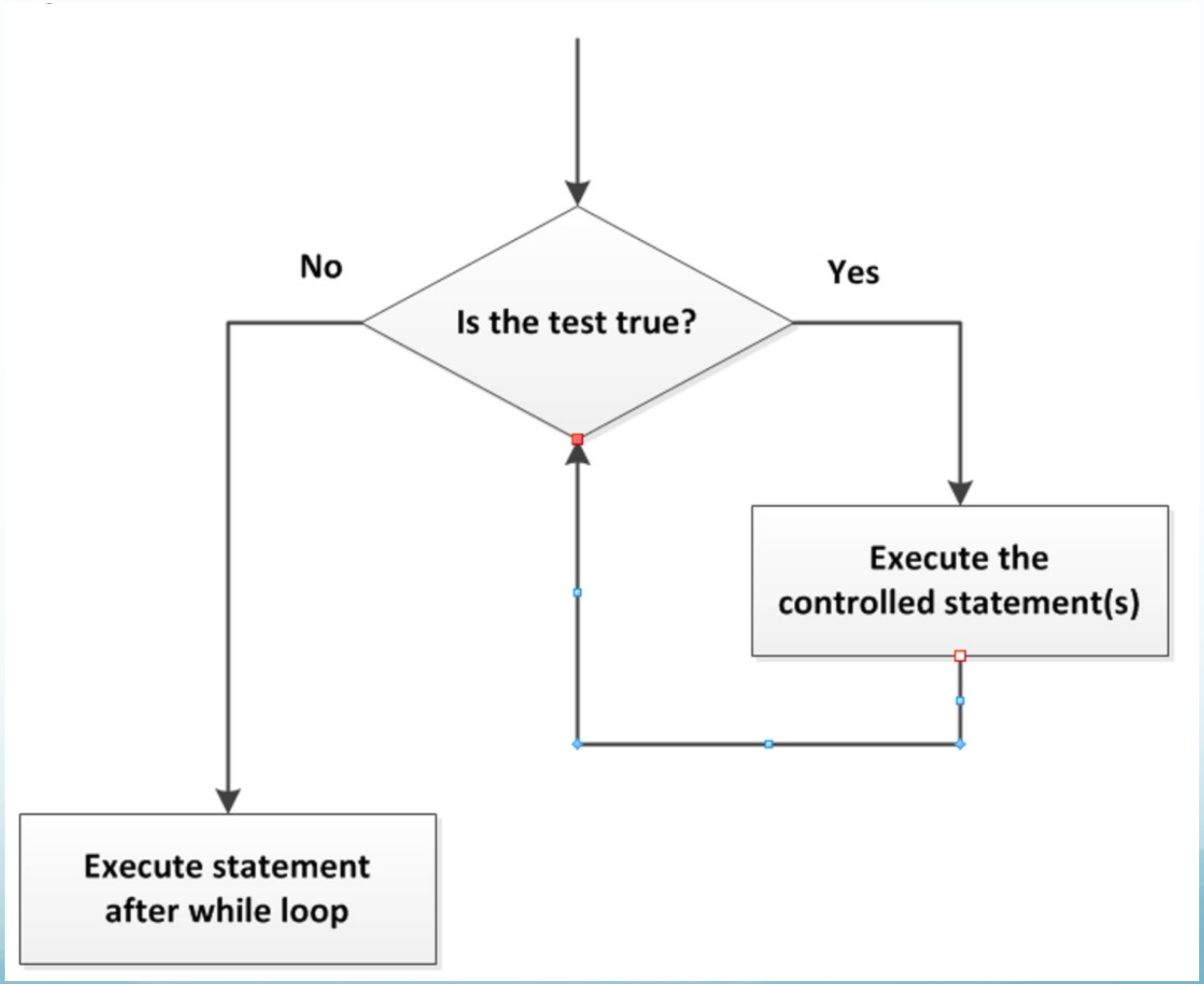
Indefinite Loops

- Sometimes, when reading from a file, or when getting user input the amount of input to be entered is not known by your program. For example, a program that asks for any number of integers to be input by the user and are then added will require some type of loop construct that runs indefinitely (as compared to definite loops which we will discuss later).
- The **while** loop is one such is one such indefinite loop and has the following general syntax:

```
while (<test>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

Indefinite Loops (continued)

- Diagram:



Indefinite Loops (continued)

- An example is as follows:

```
int number = 1;
int max = 10;
while (number <= max) {
      System.out.println("Hi there");
      number++;
}
```

Indefinite Loops (continued)

- Can create an **infinite** while loop, and in which case make sure to use the **break** statement to exit the loop:

```
int number = 1;
int max = 10;
while (true) {
      System.out.println("Hi there");
      number++;
      if (number>max) {
            break;
      }
}
```

Commenting Styles

- Proper commenting of your Java programs makes them much more readable, easier to follow and easier to maintain in the future.
- Commenting style and conventions are important and should become *second nature*.
- Proper commenting conventions must include comment *tombstones* before each class, each method and optionally before groups of like statements (e.g. constants, class variables, etc.).
- Adopting an appropriate style, like this one based on Javadoc is important.
- **IMPORTANT:** Going forward, all future labs and assignments that are submitted must adhere to a commenting style and convention like the one shown.