Text Processing, do/while, Fencepost Algorithms, boolean Type, User Errors and Assertions

CSCI 161 – Introduction to Programming I

Professor Thomas Rogers

Overview

- Reading: Chapter 5 - Program Logic and Indefinite Loops
- Topics:
  - Text Processing
  - do/while
  - Fencepost Algorithms
  - boolean Type
  - User Errors and Assertions

Text Processing

- **Text Processing** - Editing and formatting strings of text.
- **The char type** - Primitive data type *char* represents a single character of text:

```
char ch = 'A';
```

Text Processing (continued)

- Differences between char and String

	char	**String**
Type of value	primitive	object
Memory usage	2 bytes	depends on length
Methods	none	length, toUpperCase, ...
Number of letters	exactly 1	0 to many
Surrounded by	apostrophes: 'c'	quotes: "Cat"
Comparing	<, >=, ==,...	equals

Text Processing (continued)

- Can declare and assign a ***char*** variable to an escape sequence:

```
char newline = '\n';
char tab = '\t';
char quote = '\"';
```

- Values of type *char* are stored internally as 16-bit integers using a standard encoding scheme called Unicode.

Text Processing (continued)

- Java automatically converts a value of type *char* into an *int* whenever it is expecting an integer.

```
char letter = 'a' + 2; // stores ‘c’
```

  - **Note:** 'a' is Unicode value 97. Thus 2 more is 99, or 'c’.

- Can also convert the other way, but requires a cast:

```
int code = 66;
char grade = (char) code; // stores 'B'
```

Text Processing (continued)

- **Cumulative Text Algorithms** - Often need to examine a string character by character.
- For example, count the number of times a given character is in a string:

```
public static int count(String text, char c) {
    int found = 0;
    for (int i = 0; i < text.length(); i++) {
       if (text.charAt(i) == c) {
           found++;
       }
    }
    return found;
}
```

Text Processing (continued)

- **Character class** - Contains many static methods that accept a *char* parameter.

- Methods include:
  - **getNumericValue(ch)** - Converts passed in character that is a digit into a number (e.g. '6' returns 6).
  - **isDigit(ch)** - Returns a boolean indicating if the character passed in is a digit.
  - **isLetter(ch)** - Returns a boolean indicating if the character passed in is a letter ('a' - 'z' or 'A' - 'Z').
  - **isLowerCase(ch)** - Returns a boolean indicating if the character passed in is lowercase.
  - **isUpperCase(ch)** - Returns a boolean indicating if the character passed in is uppercase.
  - **toLowerCase(ch)** - Returns the lowercase version of the passed in character.
  - **toUpperCase(ch)** - Returns the uppercase version of the passed in character.

Text Processing (continued)

- **System.out.printf** - Used similarly to print and println but provides much more flexibility in formatting (the "f" stands for formatting).
  - Syntax:

```
System.out.printf(<format string>,
                  <parameters>,
                  ...,
                  <parameters>);
```

  - **format string** - Like a normal string, but contains placeholders called *format specifiers* that indicate a location where a variables value should be inserted along with the format to use.
  - **parameters** – Replacement variables, values, expressions that are used to ”fill in” specifiers within the format string.

Text Processing (continued)

- **format specifiers** - Begin with a % sign and end with a letter specifying the type of value, such as **d** for integers, **f** for floating-point numbers (real numbers of type double).

- **Common Format Specifiers:**
  - **%d** - Integer
  - **%8d** - Integer, right-aligned, 8-space-wide field
  - **%-6d** - Intever, left-aligned, 6-space-wide field
  - **%f** - Floating-point number
  - **%12f** - Floating-point number, right-aligned, 12-space-wide field
  - **%.2f** - Floating-point number, rounded to nearest hundredth (aka 2 decimal points)

Text Processing (continued)

- **Common Format Specifiers (continued):**
  - **%16.3f** - Floating-point number, rounded to nearest thousandth, 16-space-wide field
  - **%s** - String
  - **%8s** - String, right-aligned, 8-space-wide field
  - **%-9s** - String, left-aligned, 9-space-wide field
  - **%c** - character
  - **%3c** - character, right-aligned, 3-space-wide field
  - **%-4c** - character, left-aligned, 4-space-wide field

Text Processing (continued)

- **printf exercise** - Variables *color1*, *color2*,... through *color6* have names of colors. Print the names out in columns like so:

```
red       yellow      green
purple    pink        orange
```

```
System.out.printf(
    "%10s %10s %10s\n%10s %10s %10s\n",
    color1, color2, color3, color4,
    color5, color6);
```

do/while

- **do/while** - a variation of the while loop.
- Useful in situations in which you know your program needs to execute a loop at least once.
- Syntax:

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

Fencepost Algorithms

- **Fencepost algorithm** - A common programming problem that requires a kind of loop known as a fencepost loop because the problem requires actions/items at the beginning and end of the loop.
- Consider a fence: posts need to be at the beginning and end with wire in between.

  Bad - End up with trailing wire and no last post:

```
for (the length of the fence) {
    plant a post.
    attach some wire.
}
```

Fencepost Algorithms (continued)

- ***Better*** - Note the reversal (re-ordering) of the actions:

```
plant a post.
for (the length of the fence) {
    attach some wire.
    plant a post.
}
```

Fencepost Algorithms (continued)

- Consider the need for a loop that writes out 10 numbers separated by commas, as so:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

The code - note the printing of first item outside the loop then *second* action first inside the loop, and change in starting *i* value:

```
System.out.print(1); // plant post
for (int i = 2; i <=10; i++) {
    System.out.print(", "); // attach wire
    System.out.print(i); // plant post
}
```

Fencepost Algorithms (continued)

- **Variation: Fencepost with if** - An alternative to the fencepost in which the first post is not planted before the loop, but within, and then wire attached conditionally.
- Pseudocode:

```
for (the length of the fence) {
    plant a post.
    if (this isn't the last post) {
        attach some wire.
    }
}
```

Fencepost Algorithms (continued)

- **Consider the previous problem -** outputting:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

- Code:

```
for (int i = 1; i <=10; i++) {
    System.out.print(i); // plant post
    if (i != 10) {
       System.out.print(", "); // attach wire
    }
}
```

boolean type

- Named after George Boole. A primitive data type that can have the values *true* or *false*.
- The basic logical flow of algorithms in Computer Science rely on booleans.
- if/else conditionals, for and while loops are each controlled by expressions that specify a test and that test results in true or false - boolean values.

boolean type (continued)

- Logical Operators:

Operator	Meaning	Example	Value
&&	AND (conjunction)	(2 == 2) && (3 < 4)	true
||	OR (disjunction)	(1 < 2) || (2 == 3)	true
!	NOT (negation)	!(2 == 2)	false

boolean type (continued)

- Truth Table for NOT (!):

P	!p
true	false
false	true

boolean type (continued)

- Truth Table for AND (&&)

p	q	p && q
true	true	true
true	false	false
false	true	false
false	false	false

boolean type (continued)

- Truth Table for OR (||):

p	q	p || q
true	true	true
true	false	true
false	true	true
false	false	false

boolean type (continued)

- Java Operator Precedence (with logical operators):

Description	Operator
unary operators	!, ++, --, + (positive), - (negative)
multiplicative operators	*, /, %
additive operators	+, -
relational operators	<, >, <=, >=
equality operators	==, !=
logical AND	&&
logical OR	||
assignment opertors	=, +=, -=, *=, /=, %=, &&=, ||=

boolean type (continued)

- **Short-Circuited Evaluation** - The property of the logical operators && and || that prevents the second (and subsequent) operator from being evaluated if the overall result is obvious from the value of the first operand.
  - Consider these two simple rules:
    - If the current evaluation is **true** and the remaining logical operators are **OR (||)** then the overall expression is **true**.
    - If the current evaluation is **false** and the remaining logical operators are **AND (&&)** then the overall expression is **false**.

boolean type (continued)

- **boolean Methods** - A method that returns a boolean value; usually used within your program in conditionals and to carry out program logic. See *"Boolean Zen"* section from book.
- Example: Return boolean indicating if integer is two digits and both unique:

OK:

```
public static boolean isTwoUniqueDigits(int n) {
    if (n >= 10 && n <= 99 && (n % 10 != n / 10)) {
       return true;
    } else {
       return false;
    }
}
```

boolean type (continued)

- **Better:**

```
public static boolean isTwoUniqueDigits(int n) {
    return (n >= 10 && n <= 99 &&
            (n % 10 != n / 10));
}
```

boolean type (continued)

- Negating Boolean Expressions
  - A boolean expression including && and/or || that you wish to negate (because maybe you only want to use it in a conditional when the expression is NOT true) can be expressed with the negation operator (!) or be rewritten in a simplified manner.
  - The simplification is down with two rules, known as *DeMorgan's Law*, such that when simplifying:
    - Each operand is negated: == becomes !=, < becomes >=; > becomes <=, etc.
    - Each logical operator is negated (&& becomes || and vice-versa)

boolean type (continued)

- Some practice - Simplify the following via DeMorgan's Law:

```
!( str == null || x >= str.length() )
// Not (null string object or loop counter
// greater than string length)


!( n >= 1 && n <= 9) // Not a single digit number
```

User Errors and Assertions

- **User Errors (Section 5.4)**
  - *Please read the section on your own.*

- **Assertions (Section 5.5)**
  - *Please read the section on your own.*