Object Oriented Programming

(CSC 302)

Lecture Note 3

Exceptions Handling

4th December, 2019

- How to use the three-tier architecture

(to be discussed later)

How to work with continue and break in loops

- When you code loops, you usually want them to run to completion. Occasionally, though, you may need to jump to the end of a loop and exit the loop. To do that, you can use the **break** statement. Conversely, you may need to jump to the top of a loop and continue the loop. To do that, you can use the **continue** statement.
- A break statement that exits the loop

```
while (true) {
    int random = (int) (Math.random() * 10);
    System.out.println(random);
    if (random == 7) {
        System.out.println("value found - exit loop!");
        break;
    }
}
```

- To skip the rest of the statements in the current loop and jump to the top of the current loop, you can use the continue statement.
- A continue statement that jumps to the beginning of a loop

```
for (int i = 0; i < 5; i++) {
    int random = (int) (Math.random() * 10);
    if (random > 7) {
        System.out.println("invalid value – continue loop!");
        continue;
    }
    System.out.println(random);
}
```

How to catch and Handle Exception

- To prevent your applications from crashing, you can write **try/catch** statements that handle exceptions when they occur. This is known as *exception handling*, and it plays an important role in most applications.
- How to work with exception
- An **exception** is an object that contains information about an error that has occurred. When an error occurs in a method, the method throws an exception.
- If an exception is thrown when you're testing a console application, some information about the exception, including its name and *stack trace*, is displayed at the console.
- A stack trace is a list of the methods that were called before the exception occurred. The list appears in reverse order, from the last method called to the first method called.
- All exceptions are subclasses of the **Exception** class. The Exception class represents the most general type of exception. Each successive layer of subclasses represents more specific exceptions.
- The **RuntimeException** class represents exceptions that occur at runtime. All of the exceptions described in this lesson are runtime exceptions.

- Some of the classes in the Exception hierarchy

```
Exception
 RuntimeException
            NoSuchElementException
                 InputMismatchException
        IllegalArgumentException
                  NumberFormatException
        ArithmeticException
        NullPointerException
```

- Four methods that might throw an exception

Class	Method	Exception
Scanner	nextInt()	InputMismatchException
Scanner	nextDouble()	InputMismatchException
Integer	parseInt(string)	NumberFormatException
Double	parseDouble(string)	NumberFormatException

- How to catch exceptions
  - In a try statement (or try/catch statement), you code any statements that may throw an exception in a try block. Then, you can code a catch block that handles any exceptions that may occur in the try block.
  - When an exception occurs, any remaining statements in the try block are skipped and the statements in the catch block are executed.
  - Any *variables* or *objects* that are used in both the try and catch blocks must be declared before the try and catch blocks so both the try and catch blocks can access them.
  - If you use a catch block to catch a specific type of exception, you should also **import** the **package** that contains that exception class.
  - The syntax for a simple try/catch statement
    - **try** { statements }
    - **catch** (ExceptionClass exceptionName) { statements }

- Code that catches a NumberFormatException

```
String choice = "y";
while (!choice.equalsIgnoreCase("n")) {
    // get the input from the user
    System.out.print("Enter monthly investment: ");
    double monthlyInvestment;
    try {
        String line = sc.nextLine();
        monthlyInvestment = Double.parseDouble(line);
    } catch (NumberFormatException e) {
        System.out.println("Error! Invalid number. Try again.\n");
        continue;   // jump to the top of the loop
    }
    // see if the user wants to continue
    System.out.print("Continue? (y/n): ");
    choice = sc.nextLine();
    System.out.println();
}
```

- The Console class (a utility class for working with console applications)

```
import java.util.Scanner;
public class Console {
    private static Scanner sc = new Scanner(System.in);
    public static void displayLine() {
        System.out.println();
    }
    public static void displayLine(String s) {
            System.out.println(s);
    }
    public static String getString(String prompt) {
        System.out.print(prompt);
        String s = sc.nextLine();
        return s;
}
```

```
    public static int getInt(String prompt) {
            int i = 0;
            while (true) {
                System.out.print(prompt);
                try {
                    i = Integer.parseInt(sc.nextLine());
                    break;
                } catch (NumberFormatException e) {
                     System.out.println("Error! Invalid integer. Try again.");
                 }
            }
        return i;
    } // closes the getInt method
}   // closes the main class
```

- The future value application

How to work with strings

- You can initialize a String variable by assigning a String literal or another variable that refers to a String object.
- To join (or concatenate) one string with another string or another data type, you can use the + operator.
- To append a string or another data type to a string, you can use the += operator.
- When you *join* or *append* other data types to a String object, Java automatically converts the other data types to String objects so they can be used as part of the string.
- How to compare strings
- The String class provides methods that you can use to compare strings.
- The equality operator (==) checks whether two strings refer to the same String object. It’s possible for two strings to contain the same characters, but to refer to different String objects. As a result, you should not use this operator to check whether two strings contain the same characters.

isEmpty() - Returns a true value if this string contains an empty string. This method was introduced with Java 1.6.

startsWith(String) - Returns a true value if this string starts with the specified string.

endsWith(String) - Returns a true value if this string ends with the specified string.

- How to use the isEmpty method

```
if (productCode.isEmpty()) {
        System.out.println("You must enter a product code.");
}
```

- How to use the startsWith method

```
if (productDescription.startsWith("Amazon")) {
        System.out.println("This book is a Amazon book.");
}
```

- How to use the endsWith method

```
if (productDescription.endsWith("Programming")) {
        System.out.println("This book is about programming.");
}
```

- **How to work with string indexes**
- You can use an **index** to refer to each character within a string where 0 is the first character, 1 is the second character, and so on.
- **length()** - Returns an int value for the number of characters in this string.
- **indexOf(String)** - Returns an int value for the index of the first occurrence of the specified string in this string. If the string isn't found, this method returns -1.
- **indexOf(String, startIndex)** - Returns an int value for the index of the first occurrence of the specified string starting at the specified index. If the string isn't found, this method returns -1.
- **lastIndexOf(String)** - Returns an int value for the index of the last occurrence of the specified string in this string.
- **lastIndexOf(String, startIndex)** - Returns an int value for the index of the last occurrence of the specified string in this string starting at the specified index.
- **charAt(index)** - Returns the char value at the specified index.

- How to get the length of a string

```
String productCode = "java";
int length = productCode.length();
```

- How to use the length method to check for an empty string

```
if (productCode.length() == 0){
        System.out.println("You must enter a product code.");
}
```

- Code that gets the index values for the two spaces

```
String name = "Martin Van Buren";
int index1 = name.indexOf(" ");
index2 = name.indexOf(" ", index1+1);
```

- Another way to get the index values for the two spaces

```
String name = "Martin Van Buren";
int index1 = name.lastIndexOf(" ");
int index2 = name.lastIndexOf(" ", index1-1);
```

- Code that gets the index of a string

```
String name = "Martin Van Buren";
int index = name.indexOf("Van");
```

- Code that gets the character at the specified index

```
String name = "Martin Van Buren";
char char1 = name.charAt(0);
char char2 = name.charAt(1);
char char3 = name.charAt(2);
```

- How to modify strings

  trim() - Returns a String object with any spaces removed from the beginning and end of this string.

  substring(startIndex) - Returns a String object that starts at the specified index and goes to the end of the string.

  substring(startIndex, endIndex) - Returns a String object that starts at the specified start index and goes to, but doesn't include, the end index.

- Code that removes spaces from the start and end of a string

```
String choice = " y ";
choice = choice.trim();
```

- Code that parses a first name and last name from a string

```
String name = "Muhammad Salisu";
int index = name.indexOf(" ");
String firstName = name.substring(0, index);
String lastName = name.substring(index + 1);
```

- Code that adds dashes to a credit card number

```
String ccNumber = "4012888888881881";
String part1 = ccNumber.substring(0,4);
String part2 = ccNumber.substring(4,8);
String part3 = ccNumber.substring(8,12);
String part4 = ccNumber.substring(12,16);
ccNumber = part1 + "-" + part2 + "-" + part3 + "-" + part4;
```

- Code that removes dashes from a credit card number

```
String ccNumber = "4012-8888-8888-1881";
String temp = "";
for(int i = 0; i < ccNumber.length(); i++) {
        if (ccNumber.charAt(i) != '-') {
                temp += ccNumber.charAt(i);
        }
}
ccNumber = temp;
```

How to work with the StringBuilder

- String objects are **immutable**. As a result, they can’t grow or shrink.
- **StringBuilder** objects are **mutable**, which means you can modify the characters in the string. The capacity of a StringBuilder object is automatically increased if necessary.
- The StringBuilder class was introduced with Java 5. It’s designed to replace the older **StringBuffer** class.
- The StringBuffer class has identical constructors and methods as the StringBuilder class. As a result, you can use it to accomplish the same tasks.
- The StringBuffer class isn’t as efficient as the StringBuilder class, but it is **thread-safe**. As a result, you can use it instead of the StringBuilder class whenever you need to make sure your code is thread-safe.

- The StringBuilder class

  java.lang.StringBuilder;

- Constructors of the StringBuilder class

  StringBuilder() - Creates an empty StringBuilder object with an initial capacity of 16 characters.

  StringBuilder(capacity) - Creates an empty StringBuilder object with an initial capacity of the specified number of characters.

  StringBuilder(String) - Creates a StringBuilder object that contains the specified string plus an additional capacity of 16 characters.

- Some starting methods of the StringBuilder class

  append(data) - Adds a string for the specified primitive type or object to the end of the string.

  capacity() - Returns an int value for the capacity of this StringBuilder object.

  length() - Returns an int value for the number of characters in this StringBuilder object.

- Code that creates a credit card number

```
StringBuilder ccNumber = new StringBuilder();
ccNumber.append("4012");
ccNumber.append("8888");
ccNumber.append("8888");
ccNumber.append("1881");
```

- Code that shows how capacity automatically increases

```
StringBuilder name = new StringBuilder(8);
int capacity1 = name.capacity();
name.append("Raymond R. Thomas");
int length = name.length();
int capacity2 = name.capacity();
```

- More methods of the StringBuilder class

insert(index, data) - Inserts a string for the specified primitive type or object at the specified index pushing the rest of the string back.

replace(startIndex, endIndex,String) - Replaces the characters from the start index to, but not including, the end index with the specified string.

delete(startIndex, endIndex) - Removes the substring from the start index to, but not including, the end index. This moves the rest of the string forward.

deleteCharAt(index) - Removes the character at the specified index.

setCharAt(index, character) - Replaces the character at the specified index with the specified character.

charAt(index) - Returns a char value for the character at the specified index.

substring(index) - Returns a String object for the characters starting at the specified index to the end of the string.

substring(startIndex, endIndex) - Returns a String object for the characters from the start index to, but not including, the end index.

toString() - Returns a String object for the string that's stored in the StringBuilder object.

- Code that adds dashes to a credit card number

```
ccNumber.insert(4, "-");
ccNumber.insert(9, "-");
ccNumber.insert(14, "-");
```

- Code that removes dashes from a credit card number

```
for(int i = 0; i < ccNumber.length(); i++) {
        if (ccNumber.charAt(i) == '-') {
                ccNumber.deleteCharAt(i);
                i--;
        }
}
```

- Code that parses a credit card number

```
String part1 = ccNumber.substring(0,4);
String part2 = ccNumber.substring(4,8);
String part3 = ccNumber.substring(8, 12);
String part4 = ccNumber.substring(12);
```

- The Product Lister application

- The StringUtil class

```
public class StringUtil {
    public static String pad(String s, int length) {
        if (s.length() < length) {
        // append spaces until the string is the specified length
        StringBuilder sb = new StringBuilder(s);
        while (sb.length() < length) {
            sb.append(" ");
        }
        return sb.toString();
        } else {
          // truncate the string to the specified length
          return s.substring(0, length);
        }
    }
}
```