Object Oriented Programming

(CSC 302)

Lecture Note 1

Classes, Objects & Methods

4th December, 2019

How to work with classes, objects, and methods

- The **API** for the Java SE provides a large *library* of classes that are organized into *packages*.
- All classes stored in the *java.lang* package are automatically available to all Java code.
- To use classes that aren't in the *java.lang* package, you need to code an import statement at the beginning of a class. To import one class, specify the package name followed by the class name. To import all classes in a package, specify the package name followed by an asterisk (*).
- **Common packages**

  java.lang - Classes that are fundamental to the Java language, including classes that work with primitive data types and strings.

  java.util - Utility classes, including those for getting input from the console.

  java.text - Classes that handle text, including those that format numbers and dates.
- **Common classes**

  java.lang.String

  java.lang.Integer

  java.lang.Double

  java.util.Scanner

  java.text.NumberFormat

- The syntax of the import statement For a single class

  import packagename.ClassName;

- For all classes in the package

  import packagename.*;

- How to import the Scanner class
  - For the Scanner class only

    import java.util.Scanner;

- For all classes in the java.util package

  import java.util.*;

- Code that uses the Scanner class
  - If it has been imported

    Scanner sc = new Scanner(System.in);

  - If it has not been imported

    java.util.Scanner sc = new java.util.Scanner(System.in);

- How to create an object from a class
  - When you create an **object** from a Java class, you are creating an **instance** of the class. Then, you can use the methods of the class by calling them from the object.
  - Some Java classes contain **static methods**. You can call these methods directly from the class without creating an object.
  - When you create an object from a class, the **constructor** may require one or more **arguments**. These arguments must have the required data types. If there are multiple arguments, you must code them in the correct sequence, and you must separate each argument with a comma.
  - When you call a method from an object or a class, the method may require one or more arguments. Here again, these arguments must have the required data types. If there are multiple arguments, you must code them in the correct sequence, and you must separate each argument with a comma.

- When you call a method from an object or a class, the method may return a primitive value or an object. If you want to use this data, you can assign the primitive value or object to a variable.

- How to create an object from a class

  Syntax

  ```
  new ClassName(arguments);
  ```

  Example

  ```
  Scanner sc = new Scanner(System.in);   // creates a Scanner object named sc
  ```

- How to call a method from an object Syntax

```
objectName.methodName(arguments)
```

- Example

```
String line = sc.nextLine();    // get a String object from the console
```

- How to call a static method from a class Syntax

```
ClassName.methodName(arguments)
```

- Example

```
double price = Double.parseDouble(line); // convert a String to a double
```

- How to use the Scanner class to get input

  The Scanner class

  java.util.Scanner

- How to create a Scanner object

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

- A method of the Scanner object

  nextLine() - Returns all text on the current line as a String object.

- To create a **Scanner** object that gets input from the console, specify **System.in** in the parentheses.
- When the **nextLine** method of the Scanner class runs, the application waits for the user to enter text with the keyboard. To complete the entry, the user presses the Enter key.
- Since the Scanner class is not in the *java.lang* package, you typically include an **import** statement whenever you use this class.
- How to get a String object for a product code

```
System.out.print("Enter product code: ");
String productCode = sc.nextLine();
```

- How to get a String object for a quantity

```
System.out.print("Enter quantity: ");
String quantityLine = sc.nextLine();
```

- How to get a String object for a price

```
System.out.print("Enter price: ");
String priceLine = sc.nextLine();
```

- How to convert strings to numbers
  - The **Integer** and **Double** classes provide static methods that you can use for converting values from a String object to an **int** or **double** value.
- **The Integer class**

  java.lang.Integer
- Two static methods of the Integer class

  parseInt(stringName) - Attempts to convert the String object that's supplied as an argument to an int type. If successful, it returns the int value. If unsuccessful, it throws an exception.

  toString(intName) – Converts the supplied int value to a String object and returns that string object.
- **The Double class**

  java.lang.Double

- Two static methods of the Double class

  parseDouble(stringName) - Attempts to convert the String object that's supplied as an argument to a double type. If successful, it returns the double value. If unsuccessful, it throws an exception.

  toString(doubleName) – Converts the supplied double value to a String object and returns that string object.

- How to convert a String object to an int value

  int quantity = Integer.parseInt(quantityLine);

- How to convert a String object to a double value

  double price = Double.parseDouble(priceLine);

  sqrt(number) – Returns a double value that's the square root of the double argument.

  max(a, b) – Returns the greater of two float, double, int, or long arguments.

  min(a, b) – Returns the lesser of two float, double, int, or long arguments.

  random() – Returns a random double value greater than or equal to 0.0 and less than 1.0

- **The Math class**
  - You can use the static methods of the Math class to perform common arithmetic operations.
  - When a method requires one or more arguments, you can code them between the parentheses, separating multiple arguments with commas.
  - You can cast the result to the data type that you want where necessary.
- **Common static methods of the Math class**

round(floatOrDouble) – Returns the closest long value to a double value or the closet int value to a float value. The result has no decimal places.

pow(number, power) - Returns a double value of a double argument (number) that is raised to the power of another double argument (power).

- Examples
- **The round method**

```
long result = Math.round(1.667);
int result = Math.round(1.49f);
```

- **The pow method**

```
double result = Math.pow(2, 2);
double result = Math.pow(2, 3) );
int result = (int) Math.pow(5, 2);
```

- **The sqrt method**

```
double result = Math.sqrt(20.25);
```

- **The max and min methods**

```
int x = 67;
int y = 23;
int max = Math.max(x, y);
int min = Math.min(x, y);
```

- **The random method**

```
double x = Math.random() * 100;
long result = (long) x;
```

How to convert numbers to formatted strings

- The NumberFormat class
  - java.text.NumberFormat
- Three static methods of the NumberFormat class
  - **getCurrencyInstance()** - Has the default currency format ($99,999.99).
  - **getPercentInstance**() - Has the default percent format (99%).
  - **getNumberInstance()** - Has the default number format (99,999.999).
- How to create a NumberFormat object

  NumberFormat currency = NumberFormat.getCurrencyInstance();
- Three methods of a NumberFormat object
  - **format(anyNumberType)** - Returns a String object that has the format specified by the NumberFormat object.
  - **setMinimumFractionDigits(int)** - Sets the minimum number of decimal places.
  - **setMaximumFractionDigits(int)** - Sets the maximum number of decimal places.
- When you use the **format** method, the result is automatically rounded by using a rounding technique called *half-even*. This means that the number is rounded up if the preceding digit is odd, but the extra decimal places are truncated if the preceding digit is even.

- Examples
- How to format a number as currency

```
double price = 49.5;
```

- **Without method chaining**

```
NumberFormat currency = NumberFormat.getCurrencyInstance();
String priceFormatted = currency.format(price);
```

- **With method chaining**

```
String priceFormatted = NumberFormat.getCurrencyInstance().format(price);
```

- **How to format a number as a percent**

```
double discountPercent = .2;
NumberFormat percent = NumberFormat.getPercentInstance();
String discountPercentFormatted = percent.format(discountPercent);
```

- **How to format a number with one decimal place**

```
double miles = 15341.253;
NumberFormat number = NumberFormat.getNumberInstance();
number.setMaximumFractionDigits(1);
String milesFormatted = number.format(miles);
```

How to code simple control statements

- As you write applications, you need to determine when to perform certain operations and how long to continue repetitive operations. To do that, you code control statements like the while loops and if/else statements that you'll learn about in a moment. But first, you need to learn how to compare numbers and strings.
- A Boolean expression is an expression that evaluates to either **true** or **false**. You can use the relational operators to compare two numeric operands and return a Boolean value that is either true or false.
- To test two strings for equality, you must call one of the methods of the String class. The equality operator (==) does not test two strings for equality.

- **How to compare numbers**
- Relational operators (to list the operators)
  - Code that uses relational operators

```
months == 3
years != 0
discountPercent > 2.3
i < months
subtotal >= 500
quantity <= reorderPoint
```

- Logical Operators (to list them)

  subtotal >= 250 && subtotal < 500

  timeInService <= 4 || timeInService >= 12

  isValid == true & counter++ < years / isValid == true |counter++ < years

  (subtotal >= 250 && subtotal < 500) || isValid == true

- Two methods of the String class

  equals(String) - Compares the value of the String object with a String argument and returns a true value if they are equal or a false value if they are not equal. This method makes a case-sensitive comparison.

  equalsIgnoreCase(String) - Works like the equals method but is not case-sensitive.

- **Code that uses the methods of the String class**

```
choice.equals("y")
choice.equalsIgnoreCase("y")
(!lastName.equals("Sulbasun"))
code.equalsIgnoreCase(productCode)
```

- **Two common errors when testing strings for equality**

```
choice = "y"      // this does not test for equality!
choice == "y"     // this does not test for equality!
```

- **How to code if/else statement**
  - An if/else statement, or just if statement, always contains an **if** clause. In addition, it can contain one or more **else if** clauses, and a final **else** clause.
  - Any variables that are declared within a **block** for an if/else clause have **block scope**. As a result, they can only be accessed within that block.
  - If a clause requires just one statement, you don't have to enclose the statement in braces. However, it's generally considered a good programming practice to always include braces. That way, if you decide to add more statements to a clause later, you won't accidentally introduce a bug.
- The syntax of the if/else statement

```
if (booleanExpression) {statements}
[else if (booleanExpression) {statements}] ...
[else {statements}]
```

- The switch statement
- Syntax

```
 switch(expression)
 {
     case label1:
                statements
                 break;
     [case label2:
                statements
                 break;] …
     [default:
            statements
            break;]
 }
```

- Prior to Java SE 7, the switch statement could only be used with an expression that evaluated to one these types: char, byte, short, or int.
- Starting with version 7, the switch statement can also be used with string expressions. Then, the switch statement uses the equals method of the String object to compare the strings. As a result, the string in switch statements are case-sensitive.
- The switch transfers control to the appropriate label. If control isn't transferred to one of the labels, the optional default label is used.

- How to code a while loop
    - A **while** loop executes the block of statements within its braces as long as the Boolean expression is *true*. When the expression becomes *false*, the while loop skips its block of statements so execution continues with the next statement in sequence.
    - Any variables that are declared in the block of statements for a while loop have block scope. As a result, you can only access them within that block.
    - If the Boolean expression in a while loop never becomes false, the loop never ends. Then, the application goes into an infinite loop.

- The syntax of the while loop

```
while (booleanExpression) {
        statements
}
```

- A loop that prints 1 through 4 to the console

```
int i = 1;
while (i < 5) {
        System.out.println("Loop " + i);
        i = i + 1;
}
```

- How to code do while loop
  - In a do-while loop, the condition is tested after the loop is executed.
  - Syntax

```
 do {
          statements
 } while(booleanExpression);
```

- A do-while loop that calculates a future value

```
int i = 1;
int months = 36;
do {
          futureValue = (futureValue + monthlyPayment) *
          (1 + monthlyInterestRate);
     i++;
} while(i <= months);
```

- How to code the for loop
- Syntax

```
for(initialization; booleanExpression; increment/decrement)
{
   statements;
}
```

- A for loop that calculates the future value

```
int months = 36;
for(int i = 1; i <= months; i++)
{
   futureValue = (futureValue + monthlyPayment) *
           (1 + monthlyInterestRate);
}
```

- A for loop is useful when you need to increment or decrement a counter that determines how many times the loop is executed.
- The loop ends when the Boolean expression is false
- The counter variable can be declared before the for loop so that it can be used outside the for loop.

Object Oriented Programming

How to define and use classes and methods