Object Oriented Programming

(CSC 302)

Lecture Note 2

Classes, Objects and Methods (cont.)

4th December, 2019

How to code your own classes and methods

- A **class** is a template from which *objects* are made.
- An **object** is a programming entity that contains state (data) and behavior (methods).
- Once an *instance* of a class is created, it has an **identity** (a unique address) and a **state** (the data that it stores). As an application runs, an object's state may change.
- When you store a class in a **package**, the first statement of the class must be a package statement that specifies the name of the package.
- To simply development and maintenance, many applications uses a **three-tired architecture** to separate the application's user interface, business rules, and database processing. Classes are used to implement the functions performed at each layer.
- The classes in the presentation layer control the application's user *interface*. For a *console* application, the presentation layer typically consists of a class with a main method and other classes related to console input and output. For a *GUI* application, the user interface typically consists of one class for each window (called a frame) that makes up the GUI.
- The classes in the database layer handle all the application's data processing.

- The classes in the middle layer, which is sometimes called the business rules layer, act as an interface between the classes in the presentation and database layers. Sometimes, these classes correspond to business entities, such as students, customers, products, and sometimes these classes implement business rules, such as discount or credit policies. Often, the classes in this layer are referred to as business classes, and the objects created from them are called the business objects.
- Theses classes that make up each layer are often stored in packages that can be shared among applications.
- The **fields** of a class store the data of a class. A field is a variable inside an object that makes up part of its internal state.
- A field can be an **instance** variable or a **constant**
- An instance variable stores data that's available to the entire object. In other words, an instance variable stores data that's available to an instance of a class.
- An instance variable may be a *primitive* data type, an *object* created from a Java class such as the String class, or an object created from a user-defined class such as the Student class.
- The **methods** of a class define the tasks that a class can perform. Often, these methods provide a way to work with the fields of a class.

- Typically, instance variables are declared with the **private** keyword. This helps to prevent other classes from directly accessing instance variables.
- You can declare the instance variables for a class anywhere outside the constructors and methods of the class.
- A **constructor** is a special method that can be used to create, or construct, an object from a class.
- The constructor must use the same name and capitalization as the name of the class. And no return type is allowed for a constructor.
- To code a constructor that has parameters, code a data type and name for each parameter within the parentheses that follow the class name.
- The name of a constructor combined with the parameter list forms the **signature** of the constructor. Although you can code multiple constructors for a class, each constructor must have a unique signature. This is known as **overloading** a constructor.
- You can use the keyword **this** to refer to an instance variable of the current object. This is required if an instance variable has the same name as a parameter of the constructor. Otherwise, it’s optional.
- If you don’t explicitly initialize an instance variable, Java automatically initializes it for you by setting numeric variables to zero, boolean variables to false, and variables for objects to null.

- A **default** constructor has zero parameters.
- If you don't code any constructors in your class, the Java compiler automatically creates a default constructor for your class.
- If you code a constructor that has parameters, and you don't code a default constructor, the Java compiler doesn't automatically create a default constructor for your class.
- **Encapsulation** is one of the fundamental concepts of **object-oriented programming**. This means that the class controls which of its *fields* and *methods* can be accessed by other classes. As a result, the fields in the class can be hidden from other classes, and the methods in a class can be modified or improved without changing the way that other classes use them.
- Encapsulation means hiding the implementation details of an object from the clients of the object.
- The syntax for coding constructors

```
public | private ClassName([parameterList]) {
        // the statements of the constructor
}
```

- A default constructor that explicitly initializes three instance variables

```
public Product() {
        code = "";
        description = "";
        price = 0.0;
}
```

- A constructor with three parameters

```
public Product(String code, String description, double price) {
        this.code = code;
        this.description = description;
        this.price = price;
}
```

- Another way to code the constructor shown above

```
public Product(String productCode, String productDescription, double  productPrice) {
        code = productCode;
        description = productDescription;
        price = productPrice;
}
```

How to code methods

- To allow other classes to access a method, use the ***public*** keyword. To prevent other classes from accessing a method, use the ***private*** keyword.
- When you name a method, you should start each name with a verb. It's a Java standard to use the verb set for methods that ***set*** the values of instance variables and to use the verb ***get*** for methods that return the values of instance variables. These methods are typically referred to as set and get ***accessors***.
- To code a method that doesn't return data, use the ***void*** keyword for the return type.
- To code a method that returns data, code a return type in the method declaration.
- Then, code a ***return*** statement in the body of the method to return the data.
- To code a method that has a parameter, code the data type and name of the parameter between the parentheses of the method.
- **The syntax for coding a method**

```
 public | private returnType methodName([parameterList]) {
         // the statements of the method
 }
```

- A method that doesn’t accept parameters or return data

```
public void printToConsole() {
        System.out.println(code + "|" + description + "|" + price);
}
```

- A get method that returns a string

```
public String getCode() {
        return code;
}
```

- A get method that returns a double value

```
public double getPrice() {
        return price;
}
```

- A custom get method

```
public String getPriceFormatted() {
        NumberFormat currency = NumberFormat.getCurrencyInstance();
        String priceFormatted = currency.format(price);
        return priceFormatted;
}
```

- A set method

```
public void setCode(String code) {
        this.code = code;
}
```

- Another way to code a set method

```
public void setCode(String productCode) {
        code = productCode;
}
```

- **How to create an object from a class**
  - To create an object, you use the **new** keyword to create a new instance of a class. Each time the new keyword creates an object, Java calls the constructor for the object, which initializes the instance variables for the object and stores the object in memory.
  - After you create an object, you assign it to a variable. When you do, a reference to the object is stored in the variable. Then, you can use the variable to refer to the object. As a result, objects are known as **reference** types.
  - The variable for a reference type stores a reference to an object.
  - To send **arguments** to the constructor, code the arguments between the parentheses that follow the class name. To send more than one argument, separate the arguments with commas.
  - When you send arguments to the constructor, the arguments must be in the *sequence* called for by the constructor and they must have data types that are *compatible* with the data types of the parameters for the constructor.

- How to create an object in two statements
  - ClassName variableName = new ClassName(argumentList);
- No arguments
  - Product product = new Product();
- Three arguments
  - Product product = new Product("apple", "White Apple", 1.50);

How to call the methods of an object

- To call a method that doesn't accept arguments, type an empty set of parentheses after the method name.
- To call a method that accepts arguments, enter the arguments between the parentheses that follow the method name. Here, the data type of each argument must be compatible with the data type that's specified by the method's parameters.
- To code more than one argument, separate the arguments with commas.
- If a method returns data, you can code an assignment statement to assign the data that's returned to a variable. Here, the data type of the variable must be compatible with the return type of the method.
- The syntax for calling a method
  - objectName.methodName(argumentList)
- A statement that sends no arguments and returns nothing
  - product.printToConsole();
- A statement that sends one argument and returns nothing
  - product.setCode(productCode);

- A statement that sends no arguments and returns a double value

```
double price = product.getPrice();
```

- A statement that sends one argument and returns a String object

```
String priceFormatted = product.getPriceFormatted();
```

- A statement that calls a method within an expression

```
String message = "Code: " + product.getCode() + "\n";
```

How to code and call static fields and methods

- As you learned earlier in this lesson, *instance variables* and *regular methods* belong to an *object* that's created from a class. In contrast, **static fields** and **static methods** belong to the class itself. As a result, they're sometimes called **class fields** and **class methods**. To code static fields and static methods, you can use the static keyword
- When you code a static method, you can only use static fields and fields that are defined in the method. In other words, you can't use instance variables in a static method.
- To call a static method, type the name of the class, followed by the **dot** operator, followed by the name of the static method, followed by a set of parentheses. If the method requires arguments, code the arguments within the parentheses, separating multiple arguments with commas.
- To call a static field, type the name of the class, followed by the dot operator, followed by the name of the static field.

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

- **How to call static methods Syntax**
  - ClassName.methodName(argumentList)
- A static method of the Console class
  - String productCode = Console.getString("Enter the product code: ");
- **How to call static fields Syntax**
  - ClassName.fieldName
- **A static field of the Console class**
  - Console.message = "This is a test.";
  - String message = Console.message;

- Reference types compared to primitive types
- A variable for a primitive type always stores its own copy of the primitive value. As a result, changing the value for one primitive type variable doesn't change the value of any other primitive type variables.
- A variable for a reference type stores a reference to the object. This allows multiple reference type variables to refer to the same object. As a result, changing the data for one object also changes the data for any other variables that refer to that object.
- When you code a method that has a primitive type parameter, the parameter gets its own copy of the value that's passed to the method. As a result, if the method changes the value of the parameter, that change doesn't affect any variables outside the method.
- When you code a method that has a reference type parameter, the parameter refers to the object that's passed to the method. As a result, if the method uses the parameter to change the data in the object, these changes are reflected by any other variables outside the method that refer to the same object.

- **How assignment statements work**
  - For primitive types

```
double p1 = 54.50;
double p2 = p1;     // p1 and p2 store copies of 54.50
p2 = 57.50;         // only changes p2
```

  - For reference types

```
Product p1 = new Product("mysql", “Learn the basics of MySQL", 74.50);
Product p2 = p1;          // p1 and p2 refer to the same object
p2.setPrice(57.50);     // changes p1 and p2
```

- **How parameters work**
  - For primitive types

```
public static double increasePrice(double price) {
        // the price parameter is a copy of the double value
        price = price * 1.1;     // does not change price in calling code
        return price;            // returns changed price to calling code
}
```

- For reference types

```
public static void increasePrice(Product product) {
        // the product parameter refers to the Product object
        double price = product.getPrice() * 1.1;
        product.setPrice(price);            // changes price in calling code
}
```

- How method calls work
  - For primitive types

```
double price = 54.50;
price = increasePrice(price);        // assignment necessary
```

  - For reference types

```
Product product = new Product();
product.setPrice(54.50);
increasePrice(product);          // assignment not necessary
```

- **Method Overloading**
- When you create two or more methods with the same name but with different parameter lists, the methods are overloaded. It’s common to use overloaded methods to provide two or more versions of a method that work with different data types or that supply default values for omitted parameters.
- The name of the method combined with the parameter list forms the signature of the method. Although you can use the same name for multiple methods, each method must have a unique signature.
- Within a class, you can call one regular method from another regular method by coding the method name and its arguments. In other words, you don’t need to prefix the method name with the object name.
- **A signature that has one parameter**

```
public void printToConsole(String separator) {
        System.out.println(code + separator + description + separator + price);
}
```

- **A signature that doesn’t have any parameters**

```
public void printToConsole() {
        printToConsole("|"); // call the method in the first example
}
```

- **How to use the this keyword**
  - Java ***implicitly*** uses the **this** keyword for instance variables and methods. As a result, you don’t need to ***explicitly*** code it unless a method parameter or a variable that’s declared within a method has the same name as an instance variable. Then, you need to use the this keyword to identify the instance variable.
  - If you use the this keyword to call one constructor from another constructor in the same class, the statement that uses the this keyword must be the first statement in the constructor.
- **How to refer to instance variables of the current object**
  - Syntax

    this.variableName
  - A constructor that refers to three instance variables

```
public Product(String code, String description, double price) {
        this.code = code;
        this.description = description;
        this.price = price;
}
```

- **How to call a constructor of the current object Syntax**

```
this(argumentList);
```

- **A constructor that calls another constructor of the current object**

```
public Product() {
        this("", "", 0.0);
}
```

- **How to call a method of the current object Syntax**

```
this.methodName(argumentList)
```

- **A method that calls another method of the current object**

```
public String getPriceFormatted() {
        NumberFormat currency = NumberFormat.getCurrencyInstance();
        String priceFormatted = currency.format(this.getPrice());
        return priceFormatted;
}
```

- **How to pass the current object to a method**

```
methodName(this)
```

- A method that passes the current object to another method

```
public void printCurrentObject() {
        System.out.println(this);
}
```

- How to use the three-tier architecture