Object Oriented Programming

(CSC 302)

Lecture Note 5

Working with Inheritance

4th December, 2019

How to wok with inheritance

- **Inheritance** lets you create a new class based on an **existing** class. Then, the new class inherits the fields and methods of the existing class.
- A class that inherits from an existing class is called a **subclass** class, **child** class, or **derived**. A class that another class inherits is called a **base** class, **parent** class, or **superclass**.
- A subclass can **extend** the superclass by adding new *fields* and *methods* to the superclass. It can also **override** a method from the superclass with its own version of the method.
- **How the Object class works**
- The Object class in the **java.lang** package is the superclass for all classes. In other words, every class inherits the Object class or some other class that ultimately inherits the Object class. As a result, the methods defined by the Object class are available to all classes.
- When coding classes, it is recommended to override the **toString** method so that it returns a string that is concise, informative, and easy for a person to read.

- The Object class

  java.lang.Object

- Methods of the Object class
- **toString()** - Returns a String object containing the class name, followed by the @ symbol, followed by a hexadecimal representation of the hash code for this object. If that's not what you want, you can override this method as shown in the next figure.
- **equals(Object)** - Returns true if this variable refers to the same object as the specified variable. Otherwise, it returns false, even if both variables refer to objects that contain the same data. If that's not what you want, you can override the equals method as shown later in this lesson.
- **getClass()** - Returns a Class object that represents the type of this object.
- **clone()** - Returns a copy of this object as an Object object. Before you can use this method, the class must implement the **Cloneable** interface.
- **hashCode()** - Returns an **int** value that represents the hash code for this object.

- A typical value returned by a Product object's **toString** method

  ng.edu.fud.business.Product@15db9742
- A typical value returned by a Product object's **hashCode** method: 366712642
- **Access modifiers**
- Which base class members are inherited by a subclass class?
- The access modifiers play an important role in determining the inheritance of base class members in subclass classes. **A subclass class can inherit only what it can see**. A subclass class inherits all the non-private members of its base class. A subclass class inherits base class members with the following accessibility:
  - **private** - Available within the current class.
  - **public** - Available to classes in all packages.
  - **protected** - Available to classes in the same package and to subclasses.
  - ***no keyword coded*** - Available to classes in the same package.
- Access modifiers specify the accessibility of the members declared by a class.
- Protected members are accessible to the current class, to other classes in the same
- package, and to subclasses.
- An annotation is a standard way to provide information about your code. When you override a method, you can add the **@Override** annotation to the method.

- How to create a superclass
- To create a class that can be used as superclass, you define the fields, constructors, and methods of the class just as you would for any other class.
- You can use access specifiers to indicate whether members of a superclass are accessible to other classes.
- Inheritance enables you to reuse code that has already been defined by a class.
- Inheritance can be implemented by extending a class.
- When a subclass class defines different code for a method inherited from a base class by defining the method again, this method is treated as a special method - an **overridden** method.
- You can implement inheritance by using either a **concrete** class or an **abstract** class as base class, but there are some important differences that you should be aware of. These are discussed later in this lesson.
- The terms *base class*, *superclass*, and *parent class* are used interchangeably. Similarly, the terms *derived class*, *child class*, and *subclass* are user interchangeably.

The Product class

```
import java.text.NumberFormat;

public class Product {
    private String code;
    private String description;
    private double price;

    public Product() {

    }
   // get and set accessors for the code, description, and price
   // instance variables

   @Override

   public String toString() {

      return description;

   }

   public static int getCount() {  // create public access for the
                                   // count variable
          return count;

    }

}
```

- **How to create a subclass class**
- You can directly access fields that have public or protected access in the superclass from the subclass.
- You can extend the superclass by adding new fields and methods to the subclass.
- You can override the public and protected methods in the superclass by coding methods in the subclass that have the same *signatures* as methods in the superclass. However, you can't override private methods in the superclass because they aren't available to the subclass.
- You use the **super** keyword to call a constructor or method of the superclass. If you call a constructor of the superclass, it must be the **first statement** in the constructor of the subclass.
- The syntax for creating subclasses To declare a subclass
  public class SubclassName **extends** SuperClassName **{ }**
- To call a superclass constructor
  super(argumentList)
- To call a superclass method
  super.methodName(argumentList)

```
public class Book extends Product {
    private String author;
    public Book() {
        super();     // call constructor of Product superclass
        author = "";
        count++;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public String getAuthor() {
        return author;
    }
    @Override
    public String toString() {            // override the toString method
        return super.toString() +     // call method of Product superclass
            " by " + author;
    }
}
```

- Which base class members are not inherited by a subclass class?
- A subclass class does not inherit the following:
  - private members of the base class.
  - Base class members with default access, if the base class and subclass classes exist in separate packages.
  - Constructors of the base class. A derived class can call a base class's constructors, but it does not inherit them.
  - Apart from inheriting the properties and behaviors of its base class, a subclass class can also define additional properties and behaviors.

How polymorphism works

- **Polymorphism** is a feature of inheritance that lets you treat objects of different subclasses that are subclass from the same superclass as if they had the type of the superclass. If, for example, Book is a subclass of Product, you can treat a Book object as if it were a Product object.
- If you access a method of a superclass object and the method is overridden in the subclasses of that class, polymorphism determines which method is executed based on the object's type. For example, if you call the toString method of a Product object, the toString method of the Book class is executed if the object is a Book object.
- Three versions of the toString method
- The toString method in the Product superclass

```
public String toString() {
        return description;
}
```

The toString method in the Book class

```
public String toString() {
        return super.toString() + " by " + author;
}
```

- The toString method in the Software class

```
public String toString() {
        return super.toString() + " " + version;
}
```

- Code that uses the overridden methods

```
Book b = new Book();
b.setCode("java");
b.setDescription("Java Programming for Beginners");
b.setPrice(57.50);
b.setAuthor(“Anne Boem");
```

```
Software s = new Software();
s.setCode("netbeans");
s.setDescription("NetBeans");
s.setPrice(0.00);
s.setVersion("8.2");

Product p;
p = b;
System.out.println(p.toString());      // calls toString from the Book class
p = s;
System.out.println(p.toString());     // calls toString from the Software class
```

- How to cast objects
  - One potentially confusing aspect of using inheritance is knowing when to cast inherited objects explicitly. The basic rule is that Java can implicitly cast a subclass to its superclass, but you must use explicit casting if you want to treat a superclass object as one of its subclasses.
  - Java can implicitly cast a subclass to a superclass. As a result, you can use a subclass whenever a reference to its superclass is called for. For example, you can specify a Book object whenever a Product object is expected because Book is a subclass of Product.
  - You must explicitly cast a superclass object when a reference to one of its subclasses is required. For example, you must explicitly cast a Product object to Book if a Book object is expected. This only works if the Product object is a valid Book object. Otherwise, this throws a **ClassCastException**.
  - Casting affects the methods that are available from an object. For example, if you store a Book object in a Product variable, you can't call the setAuthor method because it's defined by the Book class, not the Product class.
  - You can use the **instanceof** operator to check if an object is an instance of a particular class.

- Objects of derive classes can be referred to using a reference variable of any of the following types:
- **Its own type** - An object of a class HRExecutive can be referred to using an object reference variable of type HRExecutive.
- **Its superclass** - If the class HRExecutive inherits the class Employee, an object of the class HRExecutive can be referred to using00 a variable of type Employee. If the class Employee inherits the class Person, an object of the class HRExecutive can also be referred to using a variable of type Person.
- There are differences, however, when you try to access an object using a reference variable of its own type, its base class, or an implemented interface.

```
Book b = new Book();
b.setCode("java");
b.setDescription("Java Programming for Beginners");
b.setPrice(57.50);
b.setAuthor(“Anne Boem");
```

```
Product p = b;                  // cast Book object to a Product object
p.setDescription("Test");       // OK - method in Product class
//p.setAuthor("Test");          // not OK - method not in Product class

Book b2 = (Book) p;             // cast the Product object back to a Book object
b2.setAuthor("Test");           // OK - method in Book class

Product p2 = new Product();
Book b3 = (Book) p2;            // throws a ClassCastException because
                                // p2 is a Product object not a Book object
```

Code that checks an object's type

```
Product p = new Book();   // create a Book object
if (p instanceof Book) {
        System.out.println("This is a Book object");
}
```

How to compare objects

- How the equals method of the Object class works
  - Both variables refer to the same object

```
Product product1 = new Product();
Product product2 = product1;
if (product1.equals(product2)) // expression returns true
```

    Both variables refer to different objects that store the same data

```
Product product1 = new Product();
Product product2 = new Product();
if (product1.equals(product2)) // expression returns false
```

  - To test if two objects variables refer to the same object, you can use the **equals** method of the Object class.
  - To test if two objects store the same data, you can *override* the equals method in the subclass so it tests whether all instance variables in the two objects are equal.
  - Many classes from the Java API (such as the String class) already override the equals method to test for equality.

- How to override the equals method of the Object class
- The equals method of the Product class

```
@Override
public boolean equals(Object object) {
        if (object instanceof Product) {
        Product product2 = (Product) object;
        if (code.equals(product2.getCode()) &&
                description.equals(product2.getDescription()) &&
                price == product2.getPrice()) {
                        return true;
        }
         return false;
}
```

- The equals method of the LineItem class

```
@Override
public boolean equals(Object object) {
        if (object instanceof LineItem) {
                LineItem li = (LineItem) object;
                if (product.equals(li.getProduct()) &&
                                quantity == li.getQuantity()) {
                        return true;
                }
                return false;
}
```

- How to work with Abstract and Final classes
- You can implement inheritance by using either a **concrete** class or an **abstract** class as a superclass.
- An abstract class is a class that can be inherited by other classes but that you can't use to create an object. To declare an abstract class, code the abstract keyword in the class declaration.
- An abstract class can contain fields, constructors, and methods just like other superclasses. In addition, an abstract class can contain **abstract methods**.
- To create an abstract method, you code the abstract keyword in the method declaration and you omit the method body. Abstract methods cannot have private access. However, they may have protected or default access (no access modifier).
- When a subclass inherits an abstract class, all abstract methods in the abstract class must be overridden in the subclass. Otherwise, the subclass must also be abstract.
- An abstract class doesn't have to contain abstract methods. However, any class that contains an abstract method must be declared as abstract.
- You can use variables of an abstract superclass to refer to objects of its derived class.

- How to work with the final class
- To prevent a class from being inherited, you can create a final class by coding the final keyword in the class declaration.
- To prevent subclasses from overriding a method of a superclass, you can create a final method by coding the final keyword in the method declaration. In addition, all methods in a final class are automatically final methods.
- To prevent a method from assigning a new value to a parameter, you can code the final keyword in the parameter declaration to create a final parameter. Then, if a statement in the method tries to assign a new value to the parameter, the compiler reports an error.
- A final class

```
public final class Book extends Product {
        // all methods in the class are automatically final
}
```

- A final method

```
public final String getVersion() {
        return version;
}
```

- A final parameter

```
public void setVersion(final String version) {
        // version = "new value";    // not allowed
        this.version = version;
}
```

How to work with interfaces

- An **interface** can define one or more methods. These methods are automatically public and abstract. As a result, the interface only specifies the method signatures, not any code that implements the methods.
- A class that **implements** an interface can use any constants defined by the interface. In addition, it must provide an implementation for each abstract method defined by the interface. If it doesn't, it must be declared as abstract.
- A Product class that implements the Printable interface

```
public class Product implements Printable {
        private String code; private String description; private double price;
        public Product(String code, String description, double price) {
                this.code = code;
                this.description = description;
                this.price = price;
        }
        // get and set methods for the fields
        public void print()       // implement the Printable interface
                System.out.println(description);
        }
}
```

- Interfaces compared to abstract classes

Abstract class	Interface
Variables Constants Static variables Static constants	Static constants
Methods Static methods Abstract methods	Methods(new with Java 8) Static methods(new with Java 8) Abstract methods

Advantages of an abstract class

An abstract class can use instance variables and constants as well as static variables and constants. Interfaces can only use static constants.

An abstract class can define regular methods that contain code. Prior to Java 8, an interface can't define regular methods.

An abstract class can define static methods. Prior to Java 8, an interface can't define static methods.

Advantages of an interface

A class can only directly inherit one other class, but a class can implement multiple interfaces.

A default method of an interface works much like a regular (non-static) method of a class.

- How to code interfaces
- Declaring an interface is similar to declaring a class except that you use the interface keyword instead of the class keyword.
- In an interface, all methods are automatically declared public and abstract.
- In an interface, all fields are automatically declared public, static, and final.
- When working with an interface, you can code the public, abstract, and final keywords. However, they're optional.
- An interface that doesn't contain any methods is known as a *tagging interface*. This type of interface is typically used to identify that an object is safe for a certain type of operation such as cloning or serializing.

- An interface that contain only one abstract method is known as *functional interface*. Functional interfaces are the basis for using anonymous functions called **lambda.**
- Interface methods are implicitly abstract. To define default or static methods, you must explicitly use the keyword default or static with the method declaration in an interface. Default and static methods include their implementation in an interface.
- A static method in an interface cannot be called using a reference variable. It must be called using the interface name. However, static methods can be called using reference variable or the interface name.

- The syntax for declaring an interface

```
public interface InterfaceName {
    type CONSTANT_NAME = value;                    // static constant
    returnType methodName([parameterList]);       // abstract method
}
```

- An interface that defines one abstract method

```
public interface Printable {
    void print();
}
```

- An interface that defines three abstract methods

```
public interface ProductWriter {
    boolean add(Product p);
    boolean update(Product p);
    boolean delete(Product p);
}
```

- An interface that defines three static constants

```
public interface DepartmentConstants {
        int ADMIN = 1;
        int EDITORIAL = 2;
        int MARKETING = 3;
}
```

- A tagging interface with no members

```
public interface Serializable { }
```

- How to implement an interface
  - To declare a class that implements an interface, you use the implements keyword. Then, you provide an implementation for each method defined by the interface.
  - If you forget to implement a method that's defined by an interface that you're implementing, the compiler will issue an error message.
  - A class that implements an interface can use any constant defined by that interface.

- A class that implements two interfaces

```
public class Employee implements Printable, DepartmentConstants {
        private int department;
        private String firstName;
        private String lastName;
        public Employee(int department, String lastName, String firstName) {
                this.department = department;
                this.lastName = lastName;
                this.firstName = firstName;
}
```

```
@Override
    public void print() {
        String dept = "Unknown";
        if (department == ADMIN) {
            dept = "Administration";
        } else if (department == EDITORIAL) {
            dept = "Editorial";
        } else if (department == MARKETING) {
            dept = "Marketing";
        }
        System.out.println(firstName + " " + lastName + " (" + dept + ")");
    }
}
```

- The syntax for inheriting a class and implementing an interface

  public class SubclassName **extends** SuperclassName **implements** Interface1

  [, Interface2]...{ }

- A Book class that inherits Product and implements Printable

  public class Book **extends** Product **implements** Printable {

  // class members and methods

  }

- A class can inherit another class and also implement one or more interfaces.
- If a class inherits another class that implements an interface, the subclass automatically implements the interface. However, you can code the implements keyword in the subclass for clarity.
- If a class inherits another class that implements an interface, the subclass has access to any methods of the interface that are implemented by the superclass and can override those methods.

- **How to use an interface as a parameter**
- You can declare a parameter that's used by a method as an interface type. Then, you can pass any object that implements the interface to the parameter.
- You can also declare a variable as an interface type. Then, you can assign an instance of any object that implements the interface to the variable, and you can pass the variable as an argument to a method that accepts the interface type.
- A method that accepts a Printable object

```
private static void printMultiple(Printable p, int count) {
        for (int i = 0; i < count; i++) {
                p.print();
        }
}
```

- Code that passes a Product object to the method

```
Product product = new Product("java", "Java Programming for Novice", 57.50);
printMultiple(product, 2);
```

- Another way to pass a Product object to the method

```
Printable product = new Product("java", "Java Programming for Novice", 57.50);
printMultiple(product, 2);
```

- Code that passes an Employee object to the method

```
Employee employee = new Employee( DepartmentConstants.EDITORIAL, “Salisu", “Muhammad");
printMultiple(employee, 1);
```

- How to use inheritance with interfaces
- An interface can inherit one or more other interfaces by specifying the inherited interfaces in an extends clause.
- An interface can’t inherit a class.
- A class that implements an interface must implement all abstract methods declared by the interface as well as all abstract methods declared by any inherited interfaces unless the class is defined as abstract.
- A class that implements an interface can use any of the constants declared in the interface as well as any constants declared by any inherited interfaces.

- The syntax for declaring an interface that inherits other interfaces

```
public interface InterfaceName
    extends InterfaceName1[, InterfaceName2]... {
    // the constants and methods of the interface
}
```

- A ProductReader interface

```
public interface ProductReader {
        Product getProduct(String code);
        String getProducts();
}
```

- A ProductWriter interface

```
public interface ProductWriter {
        boolean add(Product p);
        boolean update(Product p);
        boolean delete(Product p);
}
```

- A ProductConstants interface

```
public interface ProductConstants {
        int CODE_SIZE = 10;
        int DESCRIPTION_SIZE = 34;
        int PRICE_SIZE = 10;
}
```

- A ProductDAO interface that inherits these three interfaces

```
public interface ProductDAO extends ProductReader, ProductWriter,
          ProductConstants {
      // all methods and constants are inherited
}
```

- How to work with default and static methods
- With Java 8 and later, you can add regular (non-abstract) methods to an interface. These methods are known as **default** and **static** methods.
- To add a regular method to an interface, you can begin the declaration for the method with the default keyword.
- When you code a class that implements an interface, you don’t need to implement its default methods. Instead, you can use the default methods defined in the interface in your class. However, if you want to change the functionality of a default method, you can override it.
- While overriding a default method, you must not use the keyword default. Rules for overriding default and regular methods are the same.
- If an interface defines a static method, the class that implements it can define a static method with the same name, but the method in the interface isn't related to the method defined in the class.
- Static methods in a class and the interface that it implements are not related to each other. A static method in a class doesn't hide or override the static method in the interface that it implements.

- The syntax for declaring a default method (Java 8 and later)

```
default returnType methodName([parameterList]);
```

- An interface that defines a default method

```
public interface Printable {
        default void print() {
                System.out.println(toString());
        }
}
```

- A class that uses the default method

```
public class Product implements Printable {
// This class doesn't override the print method.
// As a result, it uses the print method defined by the interface.
}
```

- A class that overrides the default method

```
public class Product implements Printable {
    @Override
    public void print() {
        System.out.println(getDescription() + "|" + getPriceFormatted());
    }
}
```

- The syntax for declaring a static method (Java 8 and later)

```
static returnType methodName([parameterList]);
```

- An interface that implements a static method

```
public interface Printer {
    static void print(Printable p) {
        p.print();
    }
}
```

- Another static Method

```
public interface Interviewer {
        static void bookConferenceRoom(LocalDateTime dateTime, int duration) {
                System.out.println("Interviewer-bookConferenceRoom");
        }
}
```

- Code that calls a static method from an interface

```
Printable product = new Product("java", "Java Programming for Novice", 57.50);
Printer.print(product);
```

- To call a static method from an interface, prefix the static method with the name of the interface.