Chulalongkorn University
International School of Engineering
Department of Computer Engineering
2140105 Computer Programming Lab.

Name _____
Student ID _____
Station No. _____
Date _____

Lab 7 – Object-Oriented Programming Concepts (Episode I).

Objectives:

Upon completing this lab, students should:

- Understand the concepts of Object-Oriented programming.
- Understand what objects, classes, interfaces, inheritance and packages are.

In Summary.

What Is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lab introduces the concept of data encapsulation and the benefits of using it in your design.

What Is a Class?

A class is a blueprint or prototype from which objects are created. The class models the state and behavior of a real-world object.

What Is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This lab shows you how classes inherit states and behaviors from their parents, and how class derives from another class.

What Is an Interface?

An interface is a contract between a class and the outside world. Interface is a set of operations/behaviors that define the method signatures without the implementations. When a class implements an interface, it guarantees the behavior specified by that interface.

What Is a Package?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage.

In Details

What Is an Object?

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: *state,* and *behavior*.

- Dogs have states (name, color, breed, hungry) and behaviors (barking, fetching, wagging tail).
- Bicycles also have states (current gear, current pedal cadence, current speed) and behaviors (changing gear, changing pedal cadence, applying brakes).

Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

For each object that you see, ask yourself two questions:

- "What possible states can this object be in?" and
- "What possible behavior can this object perform?"

Your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

Figure 1 shows a circle with an inner circle filled with items, surrounded by gray wedges representing methods that allow access to the inner circle.

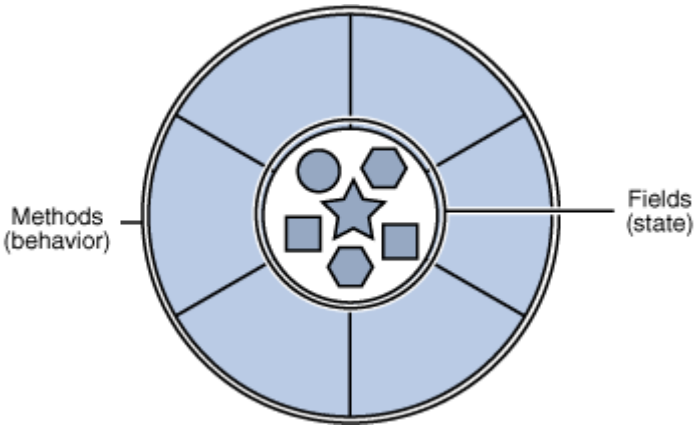

Figure 1 A software object.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages). *Methods* operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* — a fundamental principle of object-oriented programming.

Consider a bicycle in Figure 2, for example:

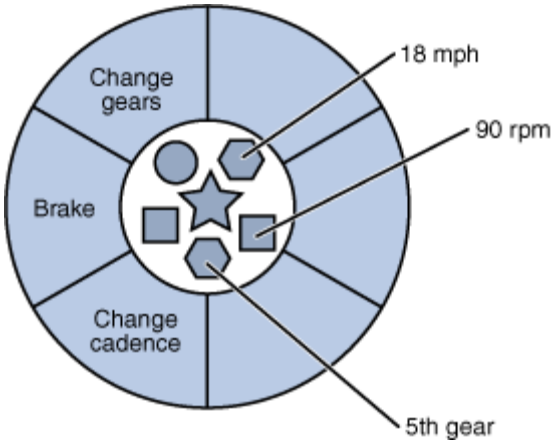

Figure 2 A bicycle modeled as a software object.

By attributing states (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

1. *Modularity*: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. *Information-hiding*: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. *Code re-use*: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. *Plug-ability and debugging ease*: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

What Is a Class?

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

Your turn

The following **`Bicycle`** class is one possible implementation of a bicycle:
Go through the following steps in developing **`Bicycle`** class.

(1) Create new Java project, named **lab7** and create a new class, **Bicycle**.

```
class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:" + cadence + " speed:" + speed +
                           " gear:" + gear);
    }
}
```

The fields **cadence**, **speed**, and **gear** represent the object's state, and the methods (**changeCadence, changeGear, speedUp** etc.) define its interaction with the outside world.

You may have noticed that the **Bicycle** class does not contain a **main** method. That's because it's not a complete application; it's just the blueprint for bicycles that might be *used* in an application. The responsibility of creating and using new **Bicycle** objects belongs to some other class in your application.

(2) Create a **BicycleDemo** class that creates two separate **Bicycle** objects and invokes their methods:

```
class BicycleDemo {
    public static void main(String[] args) {

    // Create two different Bicycle objects
    Bicycle bike1 = new Bicycle();
    Bicycle bike2 = new Bicycle();

    // Invoke methods on those objects
    bike1.changeCadence(50);
    bike1.speedUp(10);
    bike1.changeGear(2);
    bike1.printStates();

    bike2.changeCadence(50);
    bike2.speedUp(10);
    bike2.changeGear(2);
    bike2.changeCadence(40);
```

```
    bike2.speedUp(10);
    bike2.changeGear(3);
    bike2.printStates();
    }
}
```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

What Is Inheritance?

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. In this example, `Bicycle` now becomes the *superclass* of **`MountainBike`**, **`RoadBike`**, and **`TandemBike`** as shown in Figure 3. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:

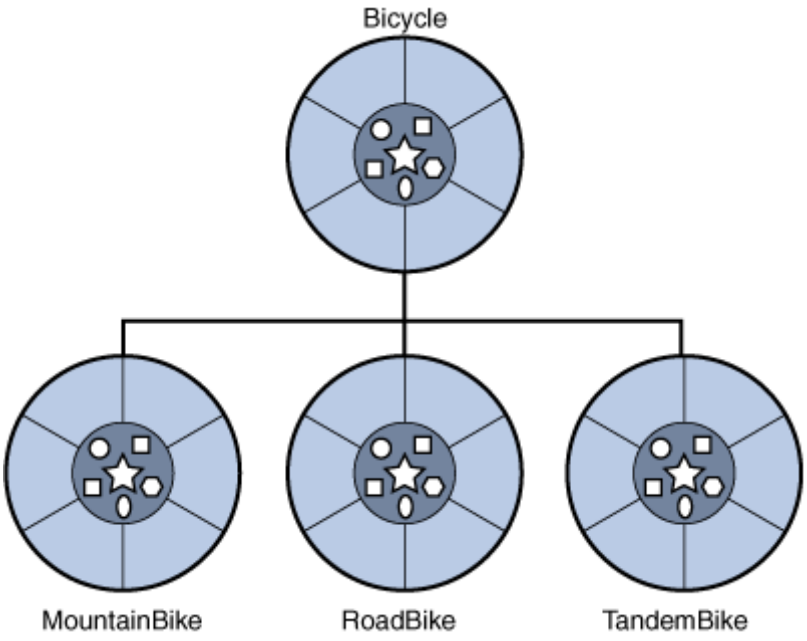

Figure 3 A hierarchy of bicycle classes.

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the **`extends`** keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {

    // new fields and methods defining a mountain bike would go here

}
```

This gives **MountainBike** all the same fields and methods as **Bicycle.** However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

What Is an Interface?

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies (they are called *abstract methods*). A bicycle's behavior, if specified as an interface, might appear as follows:

```
interface Bicycle {

    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

To implement this interface, the name of your class would change (to **ACMEBicycle**, for example), and you'd use the implements keyword in the class declaration:

```
class ACMEBicycle implements Bicycle {

    // remainder of this class implemented as before

}
```

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

What Is a Package?

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your

computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a `String` object contains state and behavior for character strings; a `File` object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a `Socket` object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

Creating and Using Packages

To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.

> **Definition**: A *package* is a grouping of related types providing access protection and name space management. Note that types refer to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred to in this lesson simply as classes and interfaces.

The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in **`java.lang`**, classes for reading and writing (input and output) are in **`java.io`**, and so on. You can put your types in packages too.

Suppose you write a group of classes that represent graphic objects, such as circles, rectangles, lines, and points. You also write an interface, **`Draggable`**, that classes implement if they can be dragged with the mouse.

```
//in the Draggable.java file
public interface Draggable {
    . . .
}

//in the Graphic.java file
public abstract class Graphic {
    . . .
}

//in the Circle.java file
public class Circle extends Graphic implements Draggable {
    . . .
}

//in the Rectangle.java file
```

```
public class Rectangle extends Graphic implements Draggable {
    . . .
}

//in the Point.java file
public class Point extends Graphic implements Draggable {
    . . .
}

//in the Line.java file
public class Line extends Graphic implements Draggable {
    . . .
}
```

You should bundle these classes and the interface in a package for several reasons, including the following:

- You and other programmers can easily determine that these types are related.
- You and other programmers know where to find types that can provide graphics-related functions.
- The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Creating a Package

To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put a package statement with that name at the top of every source file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The **`package`** statement (for example, **`package graphics;`**) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

> **Note**: If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file. For example, you can define **`public class Circle`** in the file **`Circle.java`**, define **`public interface Draggable`** in the file **`Draggable.java`**, define **`public enum Day`** in the file **`Day.java`**, and so forth.

You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be package private.

If you put the graphics interface and classes listed in the preceding section in a package called graphics, you would need six source files, like this:

```
//in the Draggable.java file
package graphics;
public interface Draggable {
    . . .
}

//in the Graphic.java file
package graphics;
public abstract class Graphic {
    . . .
}

//in the Circle.java file
package graphics;
public class Circle extends Graphic implements Draggable {
    . . .
}

//in the Rectangle.java file
package graphics;
public class Rectangle extends Graphic implements Draggable {
    . . .
}

//in the Point.java file
package graphics;
public class Point extends Graphic implements Draggable {
    . . .
}

//in the Line.java file
package graphics;
public class Line extends Graphic implements Draggable {
    . . .
}
```

If you do not use a package statement, your type ends up in an unnamed package (default package in Eclipse). Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

Naming a Package

Since Java developers are around the world, it is likely that many developers will use the same name for different types. In fact, the previous example does just that: It defines a **`Rectangle`** class when there is already a **`Rectangle`** class in the **`java.awt`** package. Still, the compiler allows both classes to have the same name if they are in different packages. The fully qualified name of each **`Rectangle`** class includes the package name. That is, the fully qualified name of the **`Rectangle`** class in the graphics package is **`graphics.Rectangle`**, and the fully qualified name of the **`Rectangle`** class in the **`java.awt`** package is **`java.awt.Rectangle`**.

Naming Conventions

Package names are written in all lowercase to avoid conflict with the names of classes or interfaces.

Companies use their reversed Internet domain name to begin their package names—for example, **`com.example.orion`** for a package named **`orion`** created by a programmer at **`example.com`**.

Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, **`com.company.region.package`**).

Packages in the Java language itself begin with **`java.`** or **`javax.`**

Using Package Members

The types that comprise a package are known as the package members.

To use a public package member from outside its package, you must do one of the following:

- Refer to the member by its fully qualified name
- Import the package member
- Import the member's entire package

Each is appropriate for different situations.

Referring to a Package Member by Its Qualified Name

So far, most of the examples in this tutorial have referred to types by their simple names, such as **`Rectangle`** and **`StackOfInts`**. You can use a package member's simple name if the code you are writing is in the same package as that member or if that member has been imported.

However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's fully qualified name, which includes the package name. Here is the fully qualified name for the **`Rectangle`** class declared in the **`graphics`** package in the previous example.

```
graphics.Rectangle
```

You could use this qualified name to create an instance of **`graphics.Rectangle`**:

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

Qualified names are all right for infrequent use. When a name is used repetitively, however, typing the name repeatedly becomes tedious and the code becomes difficult to read. As an alternative, you can import the member or its package and then use its simple name.

Importing a Package Member

To import a specific member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement, if there is one.

Here's how you would import the `Rectangle` class from the `graphics` package created in the previous section.

```
import graphics.Rectangle;
```

Now you can refer to the **`Rectangle`** class by its simple name.

```
Rectangle myRectangle = new Rectangle();
```

This approach works well if you use just a few members from the **`graphics`** package. But if you use many types from a package, you should import the entire package.

Importing an Entire Package

To import all the types contained in a particular package, use the import statement with the asterisk (`*`) wildcard character.

```
import graphics.*;
```

Now you can refer to any class or interface in the `graphics` package by its simple name.

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

The asterisk in the import statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all the classes in the **`graphics`** package that begin with A.

```
import graphics.A*;     //does not work
```

Instead, it generates a compiler error. With the import statement, you generally import only a single package member or an entire package.

For convenience, the Java compiler automatically imports three entire packages for each source file:

(1) the package with no name (default package),
(2) the **`java.lang`** package, and
(3) the current package (the package for the current file).

Name Ambiguities

If a member in one package shares its name with a member in another package and both packages are imported, you must refer to each member by its qualified name. For example, the `graphics` package defined a class named **`Rectangle`**. The **`java.awt`** package also contains a **`Rectangle`** class. If both **`graphics`** and **`java.awt`** have been imported, the following is ambiguous.

```
Rectangle rect;
```

In such a situation, you have to use the member's fully qualified name to indicate exactly which **`Rectangle`** class you want. For example,

```
graphics.Rectangle rect;
```

Managing Source and Class Files

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **`.java`**. For example:

```
// in the Rectangle.java file
package graphics;
public class Rectangle() {
   . . .
}
```

Then, put the source file in a directory whose name reflects the name of the package to which the type belongs:

```
.....\graphics\Rectangle.java
```

The qualified name of the package member and the path name to the file are parallel, assuming the Microsoft Windows file name separator backslash (for Unix, use the forward slash).

class name	**`graphics.Rectangle`**
pathname to file	**`graphics\Rectangle.java`**

As you should recall, by convention a company uses its reversed Internet domain name for its package names. The Example company, whose Internet domain name is **`example.com`**, would precede all its package names with **`com.example`**. Each component of the package name corresponds to a subdirectory. So, if the Example company had a **`com.example.graphics`** package that contained a **`Rectangle.java`** source file, it would be contained in a series of subdirectories like this:

```
....\com\example\graphics\Rectangle.java
```

When you compile a source file, the compiler creates a different output file for each type defined in it. The base name of the output file is the name of the type, and its extension is **`.class`**. For example, if the source file is like this

```
// in the Rectangle.java file
package com.example.graphics;
public class Rectangle{
          . . .
}

class Helper{
      . . .
}
```

then the compiled files will be located at:

```
<path to the parent directory of the output files>\com\example\graphics\Rectangle.class
<path to the parent directory of the output files>\com\example\graphics\Helper.class
```

Like the **.java** source files, the compiled **.class** files should be in a series of directories that reflect the package name. However, the path to the **.class** files does not have to be the same as the path to the **.java** source files. You can arrange your source and class directories separately, as:

<path_one>\sources\com\example\graphics\Rectangle.java

<path_two>\classes\com\example\graphics\Rectangle.class

By doing this, you can give the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, **<path_two>\classes**, is called the class path, and is set with the **CLASSPATH** system variable. Both the compiler and the JVM construct the path to your **.class** files by adding the package name to the class path. For example, if

<path_two>\classes

is your class path, and the package name is

com.example.graphics,

then the compiler and JVM look for **.class** files in

<path_two>\classes\com\example\graphics.

A class path may include several paths, separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in your class path.

In Eclipse

- When you create a project, Eclipse creates a folder under the workspace folder.
- In the project, creating a package will create a folder under the project folder. All file in the same package will be placed automatically in the same folder.
- By default, Eclipse puts all source file in **src** subfolder, and all classess in **bin** subfolder of your package folder.

Your turn

Import **`package.jar`** into your project and complete the following questions:

1. Change the **`Server.java`**, **`Client.java`** and **`Utilities.java`**, so that the destination package are as the table below

Package Name	Class Name
`mygame.server`	`Server`
`mygame.shared`	`Utilities`
`mygame.client`	`Client`

2. Go to the project folder and check the folder and its subfolder content.

References:

Lesson: Object-Oriented Programming Concepts, http://java.sun.com/docs/books/tutorial/java/concepts/index.html.

Horstmann, Cay. *Object-Oriented Design & Patterns*, Wiley, 2004.

Lab 7 Exercise

Your turn

1. Referring to Lab 5 exercise. If you have not finished it, do it now. Otherwise, copy the file(s) into package **`util`** in Lab 7.
2. Create a new class called **`Array`** in package **`util`** and create the following methods:
   - **`print(int[])`** – lists all elements in the array,
     `{ elem`$_0$**`, elem`**$_1$**`, …, elem`**$_{n-1}$ **`}`**
   - **`print(double[])`** – same as **`print(int[])`** except for double elements
   - **`print(int[][])`** – lists all elements in the two-dimensional array,
     `{ elem`$_{0,0}$**`, elem`**$_{0,1}$**`, …, elem`**$_{0,n-1}$**`,`**
     `elem`$_{1,0}$**`, elem`**$_{1,1}$**`, …, elem`**$_{1,\ n-1}$**`,`**
     `…`
     `elem`$_{m-1,0}$**`, elem`**$_{m-1,1}$**`, …, elem`**$_{m-1,n-1}$ **`}`**
   - **`print(double[][])`** – same as **`print(int[][])`** except for double elements.
3. Export to **`util.jar`** and save it for future use.
4. Write a program to test your **`Array`** utility class by creating some arrays and using method **`print`** to show the content of the arrays.

Lab 7 – Object-Oriented Programming Concepts (Episode I).

Task	Description	Result	Note
1	**`Bicycle`**.**`java`** and **`BicycleDemo.java`**		
2	**`Package.jar`**		
3	**`print(int[])`**		
4	**`print(double[])`**		
5	**`print(int[][])`**		
6	**`print(double[][])`**		
7	Array utility test program		