STUDENT LESSON

Inheritance, Polymorphism, and Abstract Classes

**INTRODUCTION:** A class represents a set of objects that share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea. However, something like this can be done in most programming languages. The central new idea in object-oriented programming is to allow classes to express the similarities among objects that share some, but not all, of their structure and behavior. Such similarities can be expressed using inheritance and polymorphism.

The key topics for this lesson are:

A. Inheritance

B. Abstract Classes

C. Polymorphism

D. Interfaces

**VOCABULARY:** ABSTRACT CONCRETE CLASS

EARLY BINDING INSTANCE

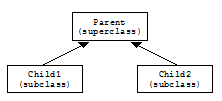
INTERFACE LATE BINDING

POLYMORPHISM SUBCLASS

SUPERCLASS

**DISCUSSION:** A. Inheritance

1. A key element in Object Oriented Programming (OOP) is the ability to derive new classes from existing classes by adding new methods and redefining existing methods. The new class can inherit many of its attributes and behaviors from the existing class. This process of deriving new classes from existing classes is called *inheritance*, which was introduced in Lesson A11, *Inheritance*.



The more general class that forms the basis for inheritance is called the *superclass*. The more specialized class that inherits from the superclass is called the *subclass* (or *derived* class).

2. In Java, all classes belong to one big hierarchy derived from the most basic class, called Object. This class provides a few features common to all objects; more importantly, it makes sure that any object is an instance of the Object class, which is useful for implementing structures that can deal with any type of object. If we start a class from “scratch,” the class automatically extends Object. For example:

**public class** SeaCreature{

...

}

is equivalent to:

**public class** SeaCreature **extends** Object{

...

}

when new classes are derived from SeaCreature, a class hierarchy is created. For example:

**public class** Fish **extends** SeaCreature{

...

}

**public class** Mermaid **extends** SeaCreature{

...

}

This results in the hierarchy shown below.

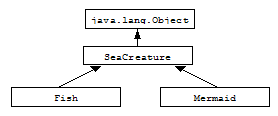


Figure 20-1. SeaCreature and two derived classes

B. Abstract Classes

1. The classes that lie closer to the top of the hierarchy are more general and abstract; the classes closer to the bottom are more specialized. Java allows us to formally define an *abstract* class. In an abstract class, some or all methods are declared **abstract** and left without code.

2. An **abstract** method has only a heading: a declaration that gives the method’s name, return type, and arguments. An **abstract** method has no code. For example, all of the methods in the definition of the SeaCreature class shown below are abstract. SeaCreature tells us what methods a sea creature must have, but not how they work.

// A type of creature in the sea

**public abstract class** SeaCreature{

// Called to move the creature in its environment

**public abstract** **void** swim();

// Attempts to breed into neighboring locations

**public abstract** **void** breed();

// Removes this creature from the environment

**public abstract** **void** die();

// Returns the name of the creature

**public abstract** String getName();

}

3. In an **abstract** class, some methods and constructors may be fully defined and have code supplied for them while other methods are **abstract**. A class may be declared abstract for other reasons as well. For example, some of the instance variables in an abstract class may belong to abstract classes.

4. More specialized subclasses of an abstract class have more and more methods defined. Eventually, down the inheritance line, the code is supplied for all methods. A class where all the methods are fully defined is called a *concrete* class. A program can only create objects of concrete classes. An object is called an *instance* of its class. An **abstract** class cannot be instantiated.

5. Different concrete classes in the same hierarchy may define the same method in different ways. For example:

**public class** Fish **extends** SeaCreature{

...

/\*\*

\* Returns the name of the creature

\*/

**public** String getName(){

**return** "Wanda the Fish";

}

...

}

**public class** Mermaid **extends** SeaCreature{

...

/\*\*

\* Returns the name of the creature

\*/

**public** String getName(){

**return** "Ariel the Mermaid";

}

...

}

C. Polymorphism

1. In addition to facilitating the re-use of code, inheritance provides a common base data type that lets us refer to objects of specific types through more generic types of references; in particular, we can mix objects of different subtypes in the same collection. For example:

SeaCreature s = **new** Fish(...);

...

s.swim();

The data type of an instance of the Fish class is a Fish, but it is also a kind of SeaCreature. Java provides the ability to refer to a specific type through more generic types of references.

2. There may be situations that require a reference to an object using its more generic supertype rather than its most specific type. One such situation is when different subtypes of objects in the same collection (array, list, etc.) are mixed. For example:

ArrayList <SeaCreature> creature = **new** ArrayList <SeaCreature>;

Creature.add( **new** Fish(...));

Creature.add( **new** Mermaid(...));

...

creature.get(currentCreature).swim();

This is possible because both Fish and Mermaid are SeaCreatures.

3. Note that the Fish and Mermaid classes provide two different implementations of the swim method. The correct method that belongs to the class of the actual object is located by the Java virtual machine. That means that one method call

String s = x.getname();

can call different methods depending on the current reference of x. When a subclass redefines the implementation of a method from its superclass, it is called overriding the method. Note that for overridden methods in Java, the actual method to call is always determined at run time. This is called *dynamic binding* or *late binding*.

4. The principle that the actual type of the object determines the method to be called is *polymorphism* (Greek for “many shapes”). The same computation works for objects of many forms and adapts itself to the nature of the objects.

5. Overloading a method is often confused with overriding a method because the names are so similar. Overloading a method means to keep the name of the method, but change the parameter list. In this case, the compiler can choose the method that is actually called because the signatures are different. The Math class has many examples of overloaded methods. For example Math.max(double a, double b) and Math.max(int a, int b) are overloaded versions of the max method. The compiler determines which one to call depending on the type of the arguments that are being passed in.

D. Interfaces

1. In Lesson A14, *Boolean Algebra*, it was learned that Java has a class-like form called an *interface* that can be used to encapsulate only abstract methods and constants. An interface can be thought of as a blueprint or design specification. A class that uses this interface is a class that *implements the* *interface*.

2. An interface is similar to an abstract class: it lists a few methods, giving their names, return types, and argument lists, but does not give any code. The difference is that an abstract class my have its constructors and some of its methods implemented, while an interface does not give any code for its methods, leaving their implementation to a class that implements the interface.

3. interface and implements are Java reserved words. Here is an example of a simple Java interface:

**public interface** Comparable{

**public int** compareTo(Object other);

}

This looks much like a class definition, except that the implementation of the compareTo() method is omitted. A class that implements the Comparable interface must provide an implementation for this method. The class can also include other methods and variables. For example,

**class** Location **implements** Comparable{

public int compareTo(Object other){

. . . // do something -- presumably, compare objects

}

. . . // other methods and variables

}

Any class that implements the Comparable interface defines a compareTo() instance method. Any object created from such a class includes a compareTo() method. We say that an object implements an interface if it belongs to a class that implements the interface. For example, any object of type Location implements the Comparable interface. Note that it is not enough for the object to include a compareTo() method. The class that it belongs to must say that it “implements” Comparable.

4. A class can implement any number of interfaces. In fact, a class can both extend another class and implement one or more interfaces. So, we can have things like:

**class** Fish **extends** SeaCreature **implements** Locatable, Eatable{

...

}

5. An interface is very much like an abstract class. It is a class that can never be used for constructing objects, but can be used as a basis for making subclasses. Even though you can't construct an object from an interface, you can declare a variable whose type is given by the interface. For example, if Locatable is an interface defined as follows

**public interface** Locatable{

Location location();

}

then if Fish and Mermaid are classes that implement Locatable, you could say

/\*\*

\* Declare a variable of type Locatable. It can refer to

\* any object that implements the Locatable interface.

\*/

Locatable nemo;

nemo = **new** Fish(); // nemo now refers to an object

// of type Fish

nemo.location(); // Calls location () method from

// class Fish

nemo = **new** Mermaid(); // Now, nemo refers to an object

// of type Mermaid

nemo.location(); // Calls location() method from

// class Mermaid

A variable of type Locatable can refer to any object of any class that implements the Locatable interface. A statement like nemo.location(), above, is legal because nemo is of type Locatable, and any Locatable object has a location() method.

6. You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are a few interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in the next few lessons, and you will see examples of interfaces in the Marine Biology Simulation, which was developed for use in AP Computer Science courses by the College Board™.

**SUMMARY/REVIEW:** The main goals of OOP are team development, software reusability, and easier program maintenance. The main OOP concepts that serve these goals are abstraction, encapsulation, inheritance, and polymorphism. In this lesson, we reviewed these key concepts and their implementation in Java. This lesson examined how Java uses classes and interfaces, inheritance hierarchies, and polymorphism to achieve the goal of better-engineered programs.

**ASSIGNMENT:** Lab Assignment A20.1, *Old MacDonald*