***Chapters 8, 9 & 10***

## *Object-Oriented Programming*

As you know, all computer programs consist of two elements: *code* and *data*. Furthermore, a program can be conceptually organized around its code or around its data. Some programs are written around “what is happening” and others are written around “who is being affected.” The 1st way is called the *process-oriented* model since these kinds of programs are characterized as a series of linear steps (i.e., code). Procedural languages such as Pascal and C employ this model.

To manage increasing complexity, the 2nd approach, called *object-oriented* programming, was conceived. Here, a program is organized around its data (ie., objects) and a set of well-defined interfaces to that data.

### Abstraction

An essential element of OOP is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. Instead, they think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car (i.e., we can ignore the details of how the engine, transmission, and braking systems work and we are therefore free to utilize the object as a whole).

A powerful way to manage abstraction is through the use of hierarchical classifications. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating and so on. Each of these subsystems is made up of more specialized units. For instance, a sound system consists of a radio, a CD player, and a tape player. The complexity of the car consists of hierarchical abstractions.

*Class* definitions in Java provide a mechanism for creating abstractions. The *data fields* capture the essential attributes of entities being represented. *Methods* provide a mechanism for manipulating those attributes.

*Encapsulation*

*Abstraction* allows the key parts of a class to be represented, but does not describe how the data associated with a class is kept or accessed. Rules must be established for how the data can be manipulated and who is allowed to do it. This control is provided through *encapsulation*.

*Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. It builds a wall around an object’s private data, allowing users of an object to access its data only through the public methods of the object’s class.

One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Consider the transmission on an auto. It encapsulates hundreds of bits of information about your engine. The user has only one method of affecting this complex encapsulation: moving the gearshift lever. The gearshift is a well-defined interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver’s perspective, they all work the same. The power of encapsulated code in Java is that everyone knows how to access it and thus can use it regardless of the implementation details.

*Encapsulation* not only protects a *class* from outsiders, but hides the details of the *class* from users of the *class* (called **information hiding**). The user of a *class* only needs to know what a *class* does and how to make it do it. The user does not need to know the coding details.

# *The Class*

In Java, the basis of *encapsulation* is the *class*. A *class* defines the structure (i.e., data) and behavior (i.e., code) that will be shared by a set of objects. When you create a class, you specify the code and data that constitutes the class. Data defined by the class is referred to as *instance variables*. Code that operates on the data is referred to as *methods*.

Since the purpose of a *class* is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked *public* or *private*. *Public* elements can be seen and accessed by external users of the class. *Private* elements can only be accessed by code that is a member of the class.

### Inheritance

*Inheritance* is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. For example, a Golden Retriever is part of the classification *dog*, which in turn is part of the *mammal* class, which is under the larger class *animal*. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. With inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. It is through inheritance that one object can become a specific instance of a more general case.

For example, if you want to describe animals in an abstract way, you would say they have some attributes such as *size*, *intelligence*, and *type of skeletal system*. Animals also have certain behavioral aspects: they *eat*, *breathe*, and *sleep*. Collectively, these attributes and behavior is the *class definition* for animals. If you want to describe a more specific class of animals, such as mammals, more specific attributes need to be added (i.e., type of teeth, mammary glands, etc.). This is known as a *subclass* of animals, where animals are referred to as mammals’ *superclass*. Since mammals are a more precise animal, they inherit all of the attributes from animals. **Therefore, a subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the rest of the code in the system**.

Think of inheritance as a means of creating a refinement of an abstraction. The entities further down the hierarchy are more complex and less general than those higher up. The entities further down the hierarchy may inherit data members (attributes) and methods from those farther up, but not vice-vesa.

Contrast this viewpoint of a hierarchy with an organizational hierarchy. A corporate hierarchy starts with the President of the company. Directors populate the next level, followed by section heads. A section head is not a director, but is subordinate to the director. This is not a **“is a”** relationship like the one between mammal and animal. In OOP, **“is a”** relationships are used to support inheritance hierarchies. Do not confuse them with other kinds of organizational schemes such as those used to define corporate structure.

Another misuse of inheritance is confusing the **“has a”** relationship with the **“is a”** relationship. For example, a mammal has fur, or a car has wheels. Think of it this way. A car **“is a”** vehicle and it **“has a”** wheel. The wheel is an attribute that (when added to car) makes a car a distinct form of vehicle. (Note that not all vehicles have wheels, for example, snowmobiles do not!) A vehicle has an engine, so by extension a car has an engine (as does the snowmobile). Cars and snowmobiles inherit the engine attribute from vehicle. However, a car is NOT an engine, it **“is a”** vehicle. Wheels and engines are parts (like instance variables) that do not inherit from each other, while cars and vehicles are classes (albeit containing instance variables). Classes can inherit from each other. And when they do, they inherit attributes (instance variables) and behavior (methods).

*Polymorphism*

*Encapsulation* and *inheritance* impose structure on object *abstractions*. *Polymorphism* provides a degree of flexibility. *Polymorphism* is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. The concept of *polymorphism* is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities.

For example, a dog's sense of smell is polymorphic. If the dog **smells** a cat, it will bark and run after it. If the dog **smells** its food, it will salivate and run to its bowl. The same **sense of smell** is at work in both situations. The difference is what is being smelled (the input data) and the action that is consequentially taken.

**Method Overriding** is a kind of polymorphism that occurs when one class extends another. By way of review, a subclass extends a superclass. The subclass can use many or all of the methods of the superclass, and it can also define its own. Sometimes the subclass needs a more refined method than the superclass. The method is said to “override” the method defined in the superclass.

For example, cars, cabs, buses and snowmobiles are all vehicles. They are extensions of a vehicle from which they inherit (or share) attributes and behavior. One of these methods defines the general rules for driving which apply for most vehicles (i.e., cabs, to cars, and buses). However, the driving rules for snowmobiles are different. So, for snowmobiles, a unique driving method is defined that takes into account, for example, a relative lack of friction. This new driving method overrides (or replaces) the driving method contained in vehicle.

Another kind of *polymorphism* is referred to as **Method Overloading**. This occurs when a method within a class is expressed in multiple forms. The difference between the methods is in the argument (or parameter) list.

To illustrate, for the **vehicle** class, instead of defining methods titled **drivingBus** and **drivingCar**, there could be two methods titled **driving**. One **driving** method would receive a **bus** parameter, the other a **car** parameter. Each method would define how each vehicle is driven. When the **driving** method is called, Java looks at the parameter being passed, and activates the correct **driving** method.

A practical example of **Method Overloading** might be a general purpose **draw** routine. With *polymorphism*, a **draw( )** routine reference could draw any collection of objects, from a box to a circle to a face. The type of object passed to **draw** is what determines which version of the **draw** routine is called. We will see other examples to *polymorphism* during the semester.

*Class Fundamentals*

Classes have been used throughout our book. Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. The words *object* and *instance* are used interchangeably.

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. The general form of a **class** definition is:

class classname {

type instance-variable1;

type instance-variable2;

…

type instance-variableN;

type methodname1(parameter-list) {

// body of method

}

type methodname2(parameter-list) {

// body of method

}

…

type methodnameN(parameter-list) {

// body of method

}

}

The variables defined within a class are called *instance variables*, the code is called *methods*. Collectively, the methods and variables are referred to as *members* of the class. The instance variables are acted upon by the methods defined for that class. Therefore, it is the methods that determine how a class’ data can be used.

Each instance of a class (i.e., object) contains its own copy of these variables. The data for one object is separate and unique from the data for another.

Java classes do not need to have a **main()** method. You only specify one if that class is the starting point for your program. Additionally, (as we’ve seen), applets don’t require a **main()** method at all.

*A Simple Class*

Last week we saw an example of a class titled Box. The main body of the class was:

**class Box {** instance variables

**double width;**

**double height;**

**double depth;**

**}**

Currently, **Box** is a class that contains three instance variables, and no methods. You can view **Box** as a new data type. You will use the name **Box** to declare objects of type **Box**. To create an object of type **Box**, you will use a statement like the following:

**Box mybox = new Box( );** creates an object (titled **mybox**) of type **Box**

To this point, Box was merely a blueprint. After this statement executes, mybox will be an instance of Box. It now has physical reality.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every Box object will contain its own copies of the instance variables **width**, **height**, and **depth**. To access these variables, you use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable:

assigns the copy of width that is contained

Within the mybox object the value of 100

mybox.width = 100;

*Example 1*

class Box {

double width;

double height;

double depth;

}

class BoxDemo {

public static void main(String args[]) {

Box mybox = new Box( );

double vol;

// assigns values to mybox's instance variables

mybox.width = 3;

mybox.height = 6;

mybox.depth = 9;

// compute volume of box

vol = mybox.width \* mybox.height \* mybox.depth;

System.out.println("Volume =" + vol);

}

}

You should call the file that contains this program **BoxDemo.java**, because the **main( )** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class into its own file, called **Box.java** and **BoxDemo.java**, respectively. To run this program you must execute **BoxDemo.class**. When you do, you will see the following output: **Volume = 162**

*Example 2*

Each object has its own copies of the instance variables. This means that if you have two Box objects, each has its own copy of length, width and height. Changes made to the instance variables on one object have no effect on the instance variables of another.

class Box {

double width;

double height;

double depth;

}

class BoxDemo2 {

public static void main(String args[]) {

Box mybox1 = new Box( );

Box mybox2 = new Box( );

double vol;

// assigns values to mybox1's instance variables

mybox1.width = 3;

mybox1.height = 6;

mybox1.depth = 9;

// assigns values to mybox2's instance variables

mybox2.width = 10;

mybox2.height = 20;

mybox2.depth = 15;

// compute volume of first box

vol = mybox1.width \* mybox1.height \* mybox1.depth;

System.out.println("Volume =" + vol);

// compute volume of second box

vol = mybox2.width \* mybox2.height \* mybox2.depth;

System.out.println("Volume =" + vol);

} }Obtaining the object of a class is a two step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object:

Statement Effect

Box mybox;

null

mybox

In the above example, mybox is declared as a reference to an object of type Box.

After this line executes, mybox contains the value null. Any attempt to use

mybox at this point will result in a run time error.

Width

Height

Depth

mybox = new Box( );

mybox

The **new** command allocates space in memory for your object. References to

**mybox** are now valid.

It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many objects as it needs, when it needs it.

Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. A class is a logical construct. An object has physical reality. An object occupies space in memory.

*Assignment Statements with Objects*

Box b1 = new Box ( );

Box b2 = b1;

Object reference variables act differently than you might expect when an assignment takes place. You might think that b1 and b2 refer to separate and distinct objects. Instead, after this fragment executes, b1 and b2 will both refer to the same object.

Width

Height

Depth

b1

b2

Therefore, any changes made to **b1** are changing **b2** (and vice versa). Although **b1** and **b2** refer to the same object, they are not linked in any other way. A subsequent change to **b1** will simply unhook **b1** from the original object without affecting the object or affecting **b2**.

*Adding Methods to Objects*

class Box {

double width;

double height;

double depth;

// a method to compute and return the volume

double volume( ) {

return width \* height \* depth;

}

}

class BoxDemo {

public static void main(String args[ ]) {

double vol;

Box first;

first = new Box( );

first.width = 3;

first.height = 6;

first.depth = 9;

System.out.println("Volume =" + first.volume( ));

}

}

class Box {

double width;

double height;

double depth;

// a method to compute and return the volume

double volume( ) {

return width \* height \* depth;

}

// a second method --- sets the dimensions of the box

void setDim (int w, int h, int d) {

width = w;

height = h;

depth = d;

}

}

class BoxDemo {

public static void main(String args[ ]) {

Box mybox1 = new Box ( );

Box mybox2 = new Box ( );

// initialize each box

mybox1.setDim (10, 20, 15);

mybox2.setDim (3, 6, 9);

// prints the volume of the first box

System.out.println("Volume =" + mybox1.volume( ));

// prints the volume of the second box

System.out.println("Volume =" + mybox2.volume( ));

}

}

In the Box class, you might wonder why volume is a method, as opposed to an instance variable. After all, if a variable is public, then we can access its value (for example, mybox1.volume). However, this would allow the user of the object to change its value as well, thereby creating potential inconsisten­cies between an object’s volume and its corresponding dimensions. By forcing the sender to use methods, we protect the data member from accidental miss-use.

*Constructors*

It can be tedious to initialize all the variables in a class each time an instance is created. A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized usable object immediately.

*Constructor*

class Box {

int width;

int height;

int depth;

Box ( ) {

System.out.println("Constructing Box");

width = 10;

height = 10;

depth = 10;

}

// a method to compute and return the volume

double volume ( ) {

return width \* height \* depth;

}

}

class BoxDemo {

public static void main(String args[ ]) {

Box mybox1 = new Box ( );

Box mybox2 = new Box ( );

// prints the volume of the first box

System.out.println("Volume =" + mybox1.volume( ));

// prints the volume of the second box

System.out.println("Volume =" + mybox2.volume( ));

}

}

**Output**

**Constructing Box**

**Constructing Box**

**Volume is 1000**

**Volume is 1000**

class Box {

*Parameterized Constructor*

int width;

int height;

int depth;

Box (int w, int h, int d ) {

width = w;

height = h;

depth = d;

}

// a method to compute and return the volume

double volume ( ) {

return width \* height \* depth;

}

}

class BoxDemo {

public static void main(String args[ ]) {

Box mybox1 = new Box ( 10, 20, 15);

Box mybox2 = new Box ( 3, 6, 9);

// prints the volume of the first box

System.out.println("Volume =" + mybox1.volume( ));

// prints the volume of the second box

System.out.println("Volume =" + mybox2.volume( ));

}

}

**Output**

**Volume is 3000**

**Volume is 162**

*this Keyword*

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the current object. **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

For example, the constructor in the previous example could have been written as follows:

Box (int w, int h, int d) {

this.width = w;

this.height = h;

this.depth = d;

}

In this example, the use of the word **this** is redundant, but perfectly correct. Inside **Box ( )**, **this** will always refer to the invoking object.

*Instance Variable Hiding*

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. However, you CAN have local variables (including formal parameters to methods) which overlap with the names of a class' instance variables. When a local variable has the same name as an instance variable, the local variable *hides* the instance variable. For example, what do you think would occur with the following code?

Box (int width, int height, int depth) {

width = width;

height = height;

depth = depth;

}

Formal Parameters

Does **this** make sense here?

Box (int width, int height, int depth) {

this.width = width;

this.height = height;

this.depth = depth;

}

The use of **this** in such a context can be confusing. Some programmers are careful not to use local variables and formal parameter names that hide instance variables. Others believe the contrary --- that it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. We will see more useful references to **this** in examples that follow.

*Garbage Collection and the finalize ( ) Method*

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically. When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object is reclaimed. Garbage collection only occurs sporadically during the execution of your program.

Sometimes an object will need to perform some action when it is destroyed. To handle such situations, Java provides a mechanism called *finalization*. *Finalization* allows you to define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a *finalizer* to a class, you simply define the **finalize( )** method. At Java run-time, this method is called whenever a corresponding object is about to be recycled.

The finalize( ) method has this general form:

protected void finalize( )

{

// finalization code goes here

}

It is important to understand that **finalize( )** is only called just prior to garbage collection. It is not immediately called when an object goes out-of-scope. Therefore, you do not know when (or if) **finalize( )** will be executed.

*Overloading Methods*

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java implements polymorphism.

When an overloaded method is invoked, Java uses the type and / or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods **must** differ in the type and / or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

// Overload test for a double parameter

double test (double a) {

System.out.println("double a:" + a);

return a \* a;

}

// Overload test for two integer parameters

void test (int a, int b) {

System.out.println("a and b:" + a + " " + b);

}

// Overload test for one integer parameter

void test (int a) {

System.out.println("a:" + a);

}

void test ( ) {

System.out.println("No Parameters");

}

**This program generates the following output:**

public static void main(String args[ ]) {

OverloadDemo ob = new OverloadDemo ( );

double result;

// call all versions to test ( )

ob.test( );

ob.test(10);

ob.test(10,20);

result = ob.test(123.2);

System.out.println("Result of ob.test(123.2):" + result);

class Overload {

}

// Demonstrate method overloading

class OverloadDemo {

}

No parameters

a: 10

a and b: 10 20

double a: 123.2

result of ob.test (123.2): 15178.2

Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm. To understand how, and the significance of polymorphism, consider the following.

In languages that do not support method overloading each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function **abs( )** returns the absolute value of an integer, **labs( )** returns the absolute value of a long integer, and **fabs( )** returns the absolute value of a floating-point value. Since C does not support overloading, each function has to have its owner name, even though all three functions do essentially the same thing. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name.

Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity.

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not (i.e., don't create a **sqr** method to return the *square* of an integer and the *square root* of a floating-point value).

*Overloading Constructors*

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the **Box** class.

class Box {

int width;

int height;

int depth;

Box (int w, int h, int d ) {

width = w;

height = h;

depth = d;

}

// a method to compute and return the volume

double volume ( ) {

return width \* height \* depth;

}

}

As you can see, the Box( ) constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box( ) constructor. For example the following statement is currently invalid:

**Box ob = new Box( );**

Since Box( ) requires three arguments, it's an error to call it without them. What if you simply wanted a box and did not care what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? By overloading the constructor, we can accomplish this goal.

class Box {

}

int width;

int height;

int depth;

Box (int w, int h, int d ) {

width = w;

height = h;

depth = d;

}

// constructor used when no dimensions specified

Box( ) {

width = -1;

height = -1;

depth = -1;

}

// constructor used when cube is created

Box(int len) {

width = height = depth = len;

}

// a method to compute and return the volume

int volume ( ) {

return width \* height \* depth;

}

Class OverloadCons {

}

public static void main (String args [ ]) {

int vol; // create boxes using the various constructors

Box mybox1 = new Box(10, 20, 15);

Box mybox2 = new Box( );

Box mycube = new Box(7);

vol = mybox1.volume( );

System.out.println("Volume of mybox1 is " + vol);

vol = mybox2.volume( );

System.out.println("Volume of mybox2 is " + vol);

vol = mycube.volume( );

System.out.println("Volume of mycube is " + vol); }

*Using Objects as Parameters*

So far we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods.

class Test {

int a, b;

}

Test (int i, int j) {

a = i;

b = j;

}

// return true if o is equal to the invoking object

boolean equals (Test o) {

if ((o.a == a) && (o.b == b)) return true;

else return false;

}

class PassOb {

public static void main(String args[ ]) {

Test ob1 = new Test(100, 22);

Test ob2 = new Test(100, 22);

Test ob3 = new Test(-1, -1);

System.out.println("ob1 == ob2:" + ob1.equals(ob2));

System.out.println("ob1 == ob3:" + ob1.equals(ob3));

}

}

**Output:**

ob1 = = ob2: true

ob1 = = ob3: false

*Another Look at Argument Passing*

Class CallByValue {

}

public static void main(String args[ ]) {

Test ob = new Test( );

int a = 15, b = 20;

System.out.println("a and b before call: " + a + " " + b);

ob.meth(a, b);

System.out.println("a and b after call: " + a + " " + b);

}

// Simple types are passed by value

class Test {

}

void meth (int i, int j) {

i \*= 2;

j /= 2;

}

**Output:**

a and b before call: 15 20

a and b after call: 15 20

// Objects are passed by reference

class Test {

int a, b;

}

Test (int i, int j) {

a = i;

b = j;

}

void meth (Test o) {

o.a \*= 2;

o.b /= 2;

}

Class CallByRef {

}

public static void main(String args[ ]) {

Test ob = new Test( 15, 20);

System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);

ob.meth(ob);

System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);

}

**Output:**

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

*REMEMBER: When a simple type is passed to a method, it is done by use of call-by-value. Objects are passed by use of call-by-reference.*

*Packages*

To this point, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine the entire Internet community arguing over who first named a class “Expresso.”) Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the *package*. The package is both a naming and a visibility control mechanism. You can define classes inside a *package* that are not accessible by code outside that *package*. You can also define class members that are only exposed to other members of the same *package*. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

To create a package, include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. While the default package is fine for short programs, it is inadequate for real applications.

package *pkg*;

Here, pkg is the name of the package. For example, the following statement creates a package called BoxPackage.

package BoxPackage;

Java uses file system directories to store packages. Therefore, the .class files for any classes you declare to be part of **BoxPackage** must be stored in a directory called **BoxPackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled statement is:

package *pkg1*[.*pkg2*[.*pkg3*]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as:

package java.awt.image;

needs to be store in **java\awt\image** on your Windows system.

#### The Import Statement

The *import* statement allows class members to be referred to by their simple names so that the class name does not have to be included. The statement:

import BoxPackage.\*;

imports the classes in package BoxPackage. As a result, the classes may be referred to by their short names (rather than their long names). For example, if Box is in the BoxPackages package, then without the import statement, one must say BoxPackages.Box. When the *import* statement is present, the reference can be directly to Box. More than one import specification may appear in a file.

*Access Control*

Java's access specifiers are public, private, and protected. They can be used as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Private | No Modifier | Protected | Public |
| Same Class | Yes | Yes | Yes | Yes |
| Same package  subclass | No | Yes | Yes | Yes |
| Same package  non-subclass | No | Yes | Yes | Yes |
| Different  package  subclass | No | No | Yes | Yes |
| Different  package  non-subclass | No | No | No | Yes |

*More on Inheritance*

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a *subclass* is a specialized version of a *superclass*. It inherits all of the instance variables and methods defined by the *superclass* and adds its own, unique elements.

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example. The following code creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

**Output:**

public static void main(String args{ }) {

A superOb = new A( );

B subOb = new B( );

// The superclass may be used by itself.

superOb.i = 10;

superOb.j = 20;

System.out.println("Contents of superOb: ");

superOb.showij( );

System.out.println( );

// The subclass has access to all public members of its superclass

subOb.i = 7;

subOb.j = 8;

subOb.k = 9;

System.out.println("Contents of subOb: ");

subOb.showij( );

subOb.showk( );

System.out.println( );

System.out.println("Sum of i, j and k in subOb:");

SubOb.sum( );

}

Class SimpleInheritance {

}

// Create a subclass by extending class A

class B extends A {

int k;

}

void sum( ) {

System.out.println("i+j+k: " + (i+j+k));

}

void showk ( ) {

System.out.println("k: " + k);

}

// A simple example of inheritance

// Create a superclass

class A {

int i, j;

}

void showij ( ) {

System.out.println("i and j: " + i + " " + j);

}

Contents of subperOb:

i and j: 10 20

Contents of subOb:

i and j: 7 8

k: 9

Sum of i, j and k in subOb:

i+j+k: 24

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why subOb can access **i** and **j** and **showij( )**. Also, inside **sum( )** , **i** and **j** can be referred to directly, as if they were part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

class Box {

Box(Box ob) { // pass object to constructor

width = ob.width;

height = ob.height;

depth = ob.depth; }

double width;

double height;

double depth;

Box ( double w, double h, double d ) {

width = w;

height = h;

depth = d;

}

// constructor used when no dimensions specified

Box( ) {

width = -1;

height = -1;

depth = -1;

}

// constructor used when cube is created

Box( double len ) {

width = height = depth = len;

}

// a method to compute and return the volume

double volume ( ) {

return width \* height \* depth;

}

}

class BoxWeight **extends** Box {

double weight;

}

BoxWeight(double w, double h, double d, double m) {

width = w;

height = h;

depth = d;

weight = m;

}

class DemoBoxWeight {

}

public static void main(String args{ }) {

BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);

BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);

double vol;

vol = mybox1.volume( ):

System.out.println("Volume of mybox1 is " + vol);

System.out.println("Weight of mybox1 is " + mybox1.weight);

System.out.println( );

vol = mybox2.volume( ):

System.out.println("Volume of mybox2 is " + vol);

System.out.println("Weight of mybox2 is " + mybox2.weight);

System.out.println( );

}

**Output:**

Volume of mybox1 is 3000

Weight of mybox1 is 34.3

Volume of mybox2 is 24

Weight of mybox2 is 0.076

**BoxWeight** inherits all of the characteristics of **Box** and adds to them the **weight** component. It is not necessary for **BoxWeight** to re-create all of the features found in **Box**. It can simply extend **Box** to meet is own purposes.

// Here, Box is extended to include color.

class ColorBox extends Box {

int color;

}

*Using super*

ColorBox (double w, double h, double d, int c) {

width = w;

height = h;

depth = d;

color = c; }

Note that the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box( )**. Not only does this duplicate code found in its superclass (which is inefficient), but also it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

A subclass can call a constructor method defined by its superclass by use of the following form of **super**: **super (*parameter-list*);**

Here, parameter-list specifies any parameters needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form: **super.member**

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

(Note: **super.super.member** is invalid, but a superclass can do **super.member**.)*Using Abstract Classes*

Sometimes you will want to create a superclass that only defines a generalized form of a method that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. As you will see as your create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass.

For example, assume that a class **figure( )** exists. One of the components of a **figure** might be its **area**. However, **area** is computed differently for different **figure**s (squares, cubes, triangles etc.). Therefore, the class **figure** can define an **abstract** class for area as follows: **abstract** double area( );

Subsequently, subclasses that extend **figure** will be required to define a method **area** that overrides (or redefines) the abstract **area** method defined in **figure**. If a subclass does not directly resolve an abstract method, it must then be resolved by a subclass of the subclass.

**Assignment for Chapter 9**

This is a group project. I will assign teams of three. You are to meet and discuss the inheritance charts on page 368 of the textbook. You must then create a similar diagram that demonstrates inheritance. You select the idea to diagram. However, it must not be one that appears in my notes or in the textbook. Prepare the diagram, demonstrating your understanding of inheritance. It must be at least as deep as those you see on page 368. You must then define attributes and behaviors for each level of a single leg of your chart.

Your team must do the following:

* Meet to discuss and develop your solution.
* Submit your design on paper, with a cover page on which each team member signs that they have met and participated on the project.
* Demonstrate your inheritance chart to the class on the date that I define in class. Your demonstration can utilize our computer and projector using applications such as PowerPoint or MS-Word. Or, it can be on paper and copies to overhead transparencies (I can make copies of your work on transparencies before class if needed). Take your demonstration seriously. The quality of your presentation can earn you extra credit.

**Assignment for Chapter 8**

Create a class **Rectangle**. The class has attributes **length** and **width**, each of which defaults to 1. It has methods that calculate the **perimeter** and the **area** of the rectangle. It has *set* and *get* methods for both **length** and **width**. The *set* methods should verify that **length** and width are each floating-point numbers larger than 0.0 and less than 20.0.

Write a “driver” program to test the rectangle class. It should instantiate the rectangle, and allows the user to set and get attributes about the **Rectangle**.

Extra Credit:

**Extend** the rectangle class adding additional functionality. Call your extended class **NewRectangle**. This class stores only the Cartesian coordinates of the four corners of the rectangle. The constructor calls a *set* method that accepts four sets of coordinates and verifies that each of these is in the first quadrant with no single *x-* or *y-*coordinate larger than 20.0. The *set* method also verifies that the supplied coordinates do, in fact, specify a rectangle. Provide methods to calculate the **length**, **width**, **perimeter**, and **area**. The length is the larger of the two dimensions. Include a predicate method **isSquare** which determines whether the rectangle is a square. Write a program to test class **NewRectangle**.

Remember, for extra credit, you are on your own. Don’t expect help from the tutors. Also, if you are able to complete only portions of this extra credit, I will give you a proportion of extra credit points.