CMS 167 Class Notes Writing Classes, part 1 ( Chapter 7 )

Instructor: J. Anderson

**Members** of a class

* fields (data)
  + instance variables
  + *static* class variables
* methods

**Principle: data encapsulation** = keep the data private—accessible only via the methods of the class

All the members of a class, both fields and methods have \_access modifier that tell “who” may use them

public can be accessed by members of \_any class \_

private can be accessed by members of \_only its class\_

protected can be accessed by members of this class and its subclasses and classes in the package to which this class belongs

none default is package (methods in same \_folder )

Syntax for defining a class:

public class ClassName

{

// class definition

}

Example: define an *Auto* class that allows us to calculate miles per gallon (mpg)

public class Auto

{

// class definition

}

**Defining data members:**

* use *private* access modifier
* can be initialized when they are declared

**instance variables have “class scope”**

**-** this means that all methods of the class can access the instance variable

**Syntax:**

public class ClassName

{

// define instance variables  
 private datatype identifier;   
 …

}

// for Auto class:

// define model (String), milesDriven(int), and   
// gallonsOfGas (double)

public class Auto

{

private String model;

private int milesDriven;

private double gallonsOfGas;

}

**Writing Methods:**

Constructor(s)

* called when an object is instantiated using *new*
* job is to initialize data members
* any instance variables not given a value by the constructor are \_
  + numbers get \_0
  + *chars* get the *\_Unicode null* character
  + object references get *\_null* (Ex: Strings )
  + *boolean*s get *\_false*
* constructor has same name as the class
* has NO return value (not even *void*)
* there can be multiple constructors (as long as the APIs differ)
* the default constructor (\_no arguments ) is supplied by Java. But it goes away if you define another constructor--of course, you can define your own default constructor.

*/\* Default constructor: initializes data to default values \*/*

*public ClassName( )  
{  
 //….give values to instance variables*

*}*

*// Auto default constructor*

*public Auto( )*

*{*

*model = "unknown";*

*milesDriven = 0;*

*gallonsOfGas = 0.0;*

*}*

*/\* Overloaded constructor: allows client to set beginning values for instance variables*

*\*/*

*public ClassName( datatype field1, datatype field2, … )  
{  
 ….*

*}*

// Auto overloaded constructor:

public Auto( String startModel, int startMilesDriven,

double startGallonsOfGas )

{

model = startModel;

if ( startMilesDriven > 0 )

milesDriven = startMilesDriven;

if ( startGallonsOfGas > 0.0 )

gallonsOfGas = startGallonsOfGas;

**Writing Methods:**

**Method header syntax:**

accessModifier returnType methodName( parameter list )

{

// method body

}

where parameter list consists of 0 to many pairs of

dataType identifier

Note: methods are usually declared to be  *\_\_public*

Note also: curly braces are \_\_required

**The *return* Statement**

Used by value-returning methods to return their value:

Syntax:

return expression;

where *expression* is the same datatype as the return data type

All paths within a value-returning method must return a value or the compiler will complain:

“Not all paths return a value”

For example:

public boolean isOdd( int number )

{

if ( number % 2 == 1 )

return true;

else

return false;

}

The *return* statement can optionally be used by *void* methods as simply: \_

**Parameter scope:**

When a method starts executing, its parameters have been defined and have the values sent from the client:

Example:

Client in *main*:

System.out.println( "Enter a value: " );

int testValue = scan.nextInt( );

if ( isOdd( testValue ) )

{

System.out.println( testValue + " is odd." );

}

When the client calls a method with a parameter,

The method receives a \_copy\_\_ of the parameter. This is called \_passing by value

|  |  |
| --- | --- |
| Client:  int testValue =10;  if ( isOdd( testValue ) )  {  System.out.println(  testValue + " is odd." );  } | Method:  public boolean isOdd( int number )  {  if ( number % 2 == 1 )  return true;  else  return false;  } |

|  |  |  |
| --- | --- | --- |
| \_10 |  | \_10 |

testValue number

Changing the parameter’s value does not change the original value in the client. They are two different variables.

13 public static void main( String [] args )   
14 {  
15   
16 int num = 10;  
17   
18 System.out.println( num );  
19   
20 testMethodInt( num );  
21   
22 System.out.println( num );  
23 }   
24   
25 public static void testMethodInt( int b )  
26 {  
27 b = 50;  
28 }

Output:

10

10

However, if the parameter is an object reference, for example, an array, the method can use that reference to change the values of the elements in the array.

Because the parameter is a \_address\_\_ to the array.

14 public static void main( String [] args )   
15 {   
16 int [] a = {0, 1, 2, 3};  
17 for ( int i = 0; i < a.length; i++ ) {  
18   
19 System.out.print( a[i] + " " );  
20 }  
21 System.out.println( );  
22   
23 testArray( a );  
24 for ( int i = 0; i < a.length; i++ ) {  
25   
26 System.out.print( a[i] + " " );  
27 }  
28 System.out.println( );  
29 }   
30   
31 public static void testArray( int [] b )  
32 {  
33 b[1] = 50;  
34 }

Output:

0 1 2 3   
0 50 2 3

**Scope for parameters:**

The parameter can be referenced only by the method for which it is defined.

When the method ends, the parameter is \_destroyed/discarded\_\_.

**Scope for Methods:** A method in a class can access:

* the instance variables of the class
* any parameters sent to the method
* any local variables defined by the method
* any other method in the class

Accessor Methods (“getters”):

* purpose is to retrieve data values
* name usually starts with “get”
* usually have no parameters
* always return a value (same data type as the instance variable)

Syntax:

public dataTypeOfInstanceVariable getInstanceVariable( )

{

return instanceVariable;

}

Example (Auto class)

public String getModel()

{

return model;

}

See Examples 7.3 and 7.4.

Mutators (“setters”)

* purpose is to allow client to change data values while maintaining valid values
* name usually start with “set”
* usually have a parameter which is same data type as instance variable
* usually *void*
* should validate the new value

Syntax:

public void **setInstanceVariable**( datatype newValue )  
 {  
 // if newValue is valid

// assign it to the instance variable

// otherwise,

// assign a default value

// or output an error message

// or don't change the value

}

Example: Auto class:

Note: To avoid duplicating code and for easier maintenance should the rules for valid values change over time, the constructor should call setter methods to initialize instance variables

public Auto( String startModel, int startMilesDriven,   
 double startGallonsOfGas )

{

}

Other methods:

* to do the jobs of the class
* if the job needs to be invoked by client, make it *public*
* if the job needs to be done only by methods in the class, make it *private*

\*\*Remember that methods can access all instance variables so they don’t need to be sent as arguments:

public double calculateMilesPerGallon( )

{

if ( gallonsOfGas == 0.0 )

{

return 0.0;

}

else

{

return milesDriven / gallonsOfGas;

}

}

See Examples 7.7 and 7.8.

See Grade.java and GradeClient.java

Suppose a method has a parameter with the same identifier as a data member? Which identifier has precedence?

public void setModel( String model )

{

model = m; //works fine

}

*Note: a method parameter with the same name as an instance variable \_\_*

*the instance variable. The parameter has* ***name precedence****.*

**Possible Fixes:**

1. Rename the parameter:

public void setModel( String m )

{

}

1. Use “*this*" which is a special implicit reference.

When a method is called, *this* is set to the object for which the method was called.

Ex: sportsCar.setModel( “Miata” );

public void setModel( String model )

*this* points to sportsCar

{

this.model = model;

//this.model is instance variable

//model is parameter

}

**The *toString* Method**

**Purpose: to return a string with the object’s data values**

When in client code, you try to output an object, the *toString* method of the object's class will be called by default.

If your class does not provide a *toString* method, then the *toString* method of the *Object* class is called which outputs the name of the class and the location of the object.

The *toString* method should return a *String* that contains all the data member values (usually labeled)

API:

public String toString( )

Example. *toString* of the *Auto* class:

public String toString ( )

{

return "The model is " + model + " and miles driven are "

+ \_milesDriven + " and gallons of gas are "

+ \_gallonsOfGas ; ;

}

To call the *toString* method:

Auto mini = new Auto( “MiniCooper”, 2000, 66.66 );

System.out.println( mini.toString( ) ); // explicit call

Or

System.out.println( mini ); // toString is implied

Snippets: Write *toString* for *Grade* class