AP Computer Science Name ___________________________ Per ____

Chapter 4 Notes – Writing Classes

4.1 Classes and Objects

Key Features of OOP

Java is object oriented programming (OOP) language. OOP is a programming concept that describes objects that are based on classes. Java supports three key features of OOP: *encapsulation*, *inheritance* and *polymorphism*.

- Encapsulation is the methodology of hiding certain elements of the implementation of a class but providing a public *interface* for the client software. This is an extension of information hiding or *abstraction*.

- Classes can inherit methods and variables from another class by using the `extends` keyword. The Java programming language permits a class to extend only one other class. This restriction is called single *inheritance*.

- An object in OOP has only one form, which is the form that is given to it when constructed. A variable, on the other hand, is *polymorphic* because it can refer to objects of different forms.

Common OOP Terms

Some common terms used in object oriented programming are:

Class - A class is a way to define new types of objects in the Java programming language. A class can be considered as a *blueprint*, which is a model of the *object* that you are describing.

Object – An object is an actual instance of a *class*. You get an object when you instantiate a class using `new`. An object is also known as an *instance*.

Attributes– An object has *state*, which is defined by the values of the *attributes*. An attribute is a data element of an *object*. An attribute stores information for an object. An attribute is also called a data member, an *instance data* or a data field.

Method – An object has *behaviors*, which are defined by the methods *associated* with that object. A method is a functional element of an object. A method is also called a *function* or a *procedure*.

Constructor – A constructor is defined as a method-like construct used to initialize or build a new object. Constructors have the same *name* as the corresponding class.

Package – A package is defined as a grouping of classes, subpackages or both.

Classes

A *class* is a blueprint of an object. It is the model or pattern from which objects are created. A class contains *data* declarations and *method* declarations. You have used several predefined classes from the Java standard class library to create objects (`String`, `Scanner`, etc.). The essence of object-oriented program development is the process of designing and writing our own classes to suit our specific needs. For example, consider the following class that simulates a coin.

```java
public class Coin                          // Class names start with uppercase and
{                                          // are marked public
   private final int HEADS = 0;
   private final int TAILS = 1;

   private int face;

   //-----------------------------------------------------------------
   //  Sets up the coin by flipping it initially.
   //-----------------------------------------------------------------
   public Coin ()
   {
      flip();
   }

   //-----------------------------------------------------------------
   //  Flips the coin by randomly choosing a face value.
   //-----------------------------------------------------------------
   public void flip ()
   {
      face = (int) (Math.random() * 2); // this.face
   }

   //-----------------------------------------------------------------
   //  Returns true if the current face of the coin is heads.
   //-----------------------------------------------------------------
   public boolean isHeads ()
   {
      return (face == HEADS);
   }

   //-----------------------------------------------------------------
   //  Returns the current face of the coin as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String faceName;
      if (face == HEADS)
         faceName = "Heads";
      else
         faceName = "Tails";

      return faceName;
   }
}
```

Objects

An object has:

- *State* - descriptive characteristics
- *Behavior* - what it can do (or what can be done to it)

Objects are created using the `new` reserved word. `Coin myCoin = new Coin();`

`myCoin` contains a reference (address) to a `Coin` object .

Multiple object reference variables can contain references to the same object. The variables are called *aliases*.

```
Coin c1 = new Coin();
Coin c2 = new Coin();

Coin c3 = c2;


c2 = c1;
```

Driver Programs

A *driver program* drives the use of other, more interesting parts of a program and are often used to test other parts of the software. The driver program usually contains the `main` method.

```
public class CountFlips
{
   //-----------------------------------------------------------------
   //  Flips a coin multiple times and counts the number of heads
   //  and tails that result.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int NUM_FLIPS = 1000;
      int heads = 0, tails = 0;

      Coin myCoin = new Coin();  // instantiate the Coin object

      for (int count=1; count <= NUM_FLIPS; count++)
      {
         myCoin.flip();

         if (myCoin.isHeads())
            heads++;
         else
            tails++;
      }

      System.out.println ("The number flips: " + NUM_FLIPS);
      System.out.println ("The number of heads: " + heads);
      System.out.println ("The number of tails: " + tails);
   }
}
```

Exercise: Complete the following class. The program should create two `Coin` objects and flip the coins until one of the coins lands on Heads three times. Print out the results of each flip of the coin and a message at the end stating the winner.

```
public class FlipRace
{
   //-----------------------------------------------------------------
   //  Flips two coins until one of them comes up heads three times
   //  in a row.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int GOAL = 3;
      int count1 = 0, count2 = 0;

      // Create two separate coin objects
      Coin coin1 = new Coin();
      Coin coin2 = new Coin();

      while (count1 < GOAL && count2 < GOAL)
      {
```

4.2 Data Scope

Data Scope

The *scope* of data is the area in a program in which that data can be used (referenced) and depends on where it is declared:

- Data declared at the *class* level are called *instance data* and can be used by all methods in that class.
- Data declared within a *method* are called *local data* and can be used only in that method.

Instance Data

The `face` variable in the `Coin` class is called *instance data* because each instance (object) of the `Coin` class has its own. Every `Coin` object will have it's own value for `face`. That's the only way two objects can have different *states*. When variables are declared at the class level, they are given a default value. `int` and `double` are assigned 0 `boolean` is assigned `false` and object reference types get `null.` Instance data may be *primitive types* or *reference types*. Instance variables can be accessed anywhere in the class.

Use `this.`*variable_name* to explicitly refer to the instance variable of the currently executing object. For example, in the `Coin` class above, `this.face` could be used in the `flip(), isHeads()` and `toString()` methods to explicitly refer to the `face` instance variable declared at the beginning of the class.

NOTE: It is good programming practice (and you will be expected on the AP Exam) to initialize all instance variables when declared or in the class constructor.

Local data

Local variables can be declared inside a *method*. The formal parameters of a method create *automatic local variables* when the method is invoked**.** When the method finishes, all local variables are destroyed (including the formal parameters)**.** Variables declared at this level DO NOT get a default value. If you try to use a variable at this level that has not been initialized, you will get a compile error.

```
... variable x might not have been initialized
```

The difference between instance and local variables

1) Instance variables are declared inside a class but not within a method.

```
public class Horse
{
    private double height = 15.2;
    private String breed;
    // more code …
}
```

2) Local variables are declared within a method.

```
public class AddThing
{
    int a;
    int b = 12;

    public int add()
    {
        int total = a + b;
        return total;
    }
}
```

3) Local variables must be initialized before use!

```
public class Foo
{
    public void go()
   {
        int x;
        int z = x + 3;
    }
}
```

4.3 Encapsulation

Encapsulation

We can take one of two views of an object:

- *Internal* - the variables the object holds and the methods that make the object useful
- *External* - the services that an object provides and how the object interacts

From the external view, an object is an *encapsulated* entity, providing a set of specific services. These services define the *interface* to the object. Remember encapsulation is the idea that every created object will change its own instance variables through *getter* and *setter* methods (also called accessors and mutators). An object should be *self-governing* in that any changes to the object's state (its variables) should be made only by that object's methods. We should make it difficult, if not impossible, to access an object's variables other than via its methods. The user, or *client* of an object can request its services, but it should not have to be aware of how those services are accomplished.

Visibility Modifiers

In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers.* A visibility modifier is a Java reserved word that specifies particular characteristics of a method or data value. We will study two visibility modifiers: `public` and `private`

- Members of a class (instance variables and methods) that are declared with *public visibility* can be accessed from anywhere.
- Members of a class (instance variables and methods) that are declared with *private visibility* can only be accessed from inside the class. All objects created from the same class can access each other's private data.

For the AP exam:

- Mark all classes as `public`.
- Mark all instance variables as `private`.
- Methods, constructors and constants (`static final` variables) are either `public` or `private`.
- Initialize all instance variables in the constructors.

Example: Write the definition of a *class* `PlayList` containing:

- An *instance variable* `title` of *type* `String` , *initialized* to the empty `String` .
- An *instance variable* `playCount` of *type* `int` , *initialized* to 0.

In addition, your `PlayList` *class* definition should provide an appropriately *named* "get" *method* and "set" *method* for each of these. No *constructor* need be defined.

4.4 Method Declarations

Method Declarations

A *method declaration* specifies the code that will be executed when the method is *invoked* (or called). When a method is invoked, the flow of control jumps to the method and executes its code. When complete, the flow returns to the place where the method was called and continues. The invocation may or may not return a value, depending on how the method is defined.

Example: Suppose the `main` method calls `method1`, `method1` calls `method6`, `method6` calls `method3` and `method2` calls `method4`. What method will resume execution when `method6` terminates?

The only required elements of a method declaration are the method's *return type*, *name*, a *pair of parentheses*, `()`, and a *body between braces*, `{}`.

More generally, method declarations have five components, in order:

1. Modifiers (optional)—such as `public` or `private`.
2. The return type—the data type of the value returned by the method, or `void` if the method does not return a value.
3. The method name—our convention is to start method names with a lowercase letter and capitalize the first letter of every word in the name after the first word. `getMyName`
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, `()`. If there are no parameters, you must use empty parentheses. The name of a parameter in the method declaration is called a *formal parameter*.
5. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

```
public int calc (int num1, int num2, String message)

{
      int sum = num1 + num2;
      System.out.println (message);
      return sum;
}
```

`sum` is local data. It is created each time the method is called, and is destroyed when the method finishes executing.

The `return` statement

The *return type* of a method indicates the type of value that the method sends back to the calling location. A method that does not return a value has a `void` return type. A *return statement* specifies the value that will be returned. The return *expression* must be consistent with the return *type*. Methods without a return statement must be a `void` method.

```
return expression;
```

Each time a method is called, the *actual parameters* (also called arguments) in the invocation are copied into the *formal parameters*:

```
...
count = 5;
int calcSum = obj.calc (25, count, "Hello");
System.out.println(calcSum);
```

```
public int calc (int num1, int num2, String message)
{
     int sum = num1 + num2;
     System.out.println (message);
     return sum;
}
```

<u>Output:</u>

Example (To call or invoke a method): Assume that `chipotle` is a reference to an object that has a method named `addSales`, that accepts two `int` arguments and returns their sum. Two `int` variables, `burritoSales` and `tacoSales`, have already been declared and initialized. Another `int` variable, `foodSales`, has already been declared. Write a statement that calls `addSales` in the `chipotle` object to compute the sum of `burritoSales` and `tacoSales` and that stores this value in `foodSales`.

Example (To define or create a method): Write the definition of a method `printStarLine`, which has no parameters and doesn't return anything. The method prints to standard output a single line (terminated by a newline) consisting of five stars.

Overloading Methods

Method overloading is the process of using the same method name for multiple methods. The *signature* of each overloaded method must be unique. The signature includes the *number, type* and *order* of the parameters. The compiler determines which version of the method is being invoked by analyzing the parameters. The *return type* of the method is NOT part of the signature. Constructors can also be overloaded. Overloaded constructors provide multiple ways to initialize a new object.

Example: What version of the method `tryMe` will be executed with the following statement:

```
result = tryMe (25, 2.43);
```

Version 1

```
double tryMe (int x)
{
   return x + .375;
}
```

Version 2

```
double tryMe (int x, double y)
{
   return x*y;
}
```

Example: Consider the following three method headers, what can you say is **true** about their method signatures?

1. `double findAnswer(String a, int b)`
2. `int findAnswer(String c, int d)`
3. `double findAnswer( int a, String b)`

Preconditions and Postconditions

A *precondition* is a condition that should be `true` when a method is called. A *postcondition* is a condition that should be `true` when a method finishes executing. These conditions are expressed in comments above the method header. They provide information to the users of the method, specifying what is expected to be `true` whenever the method is called. Both preconditions and postconditions are a kind of *assertion,* a logical statement that can be `true` or `false` which represents a programmer´s assumptions about a program. Try to write *preconditions* and *postconditions* as legal Java `boolean` expressions whenever possible. If the expressions evaluate to `false` your code should print out an error.

Example: Write a method `computeHeight` which accepts two `int` values, `feet` and `inches` and returns the height in inches. Include pre- and postconditions in your comments and ensure the preconditions are true in your method code.

4.5 Constructors

Constructors

Recall that a constructor is a special method that is called when an object is created (instantiated)**.** The constructor defines how the object is initialized. When writing a constructor, remember that:

- it has the same name as the class
- it does not return a value
- it has no return type, not even `void`
- it typically sets the initial values of instance variables. **NOTE:** You will be expected to initialize all instance variables in the constructors on the AP exam.

You cannot write two constructors that have the same number and type of parameters for the same class, because the platform would not be able to tell them apart. Doing so causes a compile error. The programmer does not have to define a constructor for a class, but if one is written, the default, *no argument*, constructor no longer exists and will have to explicitly be written.

Example: Write two constructors for a class called `Bicycle`. The class has three `int` instance variables: `gear, cadence` and `speed.` The first constructor should allow the user to pass three parameters to initialize the instance variables when the object is created. The second constructor should be a no-argument constructor and should initialize the instance variables to 1, 10 and 0 respectively.

```
Bicycle myBike = new Bicycle(30, 0, 8); // uses 3 argument constructor
Bicycle yourBike = new Bicycle(); // uses no-argument constructor
```

Example: Write the code for a `Book` class with the properties on the following page. Listed below is the client code and output.

```
.
.
Book book1 = new Book("The Cat in the Hat", "Dr. Suess", 10.00);
book1.giveDiscount(15);
System.out.printf("The new price of \"" + book1.getTitle() + "\" is $%.2f",
                 +  book1.getPrice());

System.out.println();

Book book2 = new Book("Where the Wild Things Are", "Maurice Sendak", 12.00);
book2.giveDiscount();
System.out.printf("The new price of \"" + book2.getTitle() + "\" is $%.2f",
                 +  book2.getPrice());

.
```

Output:

```
The new price of "The Cat in the Hat" is $8.50
The new price of "Where the Wild Things Are" is $10.80
```

1. A `Book` object has a title (`String`), and author (`String`) and a price (`double`).
2. Write a constructor for the `Book` class that has three parameters: a title (`String`), an author (`String`) and a price (`double`).
3. Write a method `getPrice` which returns the price of the book.
4. Write a method `getTitle` which returns the title of the book.
5. Write a method `giveDiscount` that modifies the price of the book by decreasing the price of the book by a give percentage which is passed as a parameter to the method call.
6. Overload the method `giveDiscount` to allow clients to call the method without passing a discount percentage. If no discount is passed, the book should be discounted 10%.