# CMS 3145 Week 2 Class Notes: Chapter 4

## Object Oriented Programming Concepts

**OBJECTIVES:**

* Understand the differences between classes and objects
* Understand object terminology including encapsulation, inheritance, and polymorphism
* Create your own classes
* Write constructors with multiple parameter lists
* Investigate static properties with class variables and instance variables

**OOPs Theory**

Java is an object oriented programming (OOP) language. This chapter covers OOP and the concepts of **encapsulation**, **instance** variables, and **static** fields and methods.

An **object** is an **instance** of a class; it is **encapsulated** functionality. The class is the blueprint——the form or mold, if you will—for the actual object that is **instantiated** with a class **constructor**. Class definitions allow logical design and code reuse. Three important characteristics of OOP are encapsulation (data hiding), polymorphism (chapter 11), and inheritance (chapter 12).

* **Encapsulation** allows data hiding, by “boxing” objects’ properties and methods. An object is truly encapsulated functionality. The private (or protected) methods and properties within the class definition cannot be altered; they can only be used and overridden.
  + When defining classes, the private variables used to hold a property value protect the object’s definition of the variable from view—thus *data hiding*.
  + The use of class variables, or **static** properties, as opposed to instance variables allows a property to be static among instances of an object. That is, you can define a public static property within a class that provides for storage of an accumulating attribute of all common objects of a class within a single run of a program. The definition of a static class variable within the class definition allows this.
* **Polymorphism** (literally, *many shapes)* allows us to use identical method names for similar but slightly different methods. Depending upon the object used, and the parameters passed, the appropriate method is called. Likewise, one method **constructor** definition can have many different forms, allowing for *overloaded* objects. The println() method you have been using has multiple signatures [blank, object, string, Boolean,….], allowing several forms of this method, depending upon the parameter chosen. The only requirement is that each method has a unique ‘**Parameter Signature**’. Multiple parameterized constructors are examples of polymorphism. *Overloading* allows you to rewrite a method in a subclass using the same name as a method in the base class. The subclass method of the same name takes precedence, or overrides, the base class method with that name.
* **Inheritance** allows derived classes (sometimes called sub classes) to be created from existing class definitions. A base (or Super) class upon which many subclasses can be derived, can be the heart of developing an entire vertical application (such as students in any school anywhere, or auto parts for any franchise, or book sales nationwide) very quickly, using one well-written base class. Inheritance is the essence of object-oriented programming. **To define a new derived class, use:**
  + “**extends***”* at the top of code, right after class name: **extends** ParentClass
  + “**super**()” in the class constructors to call the ParentClass constructor.

**Class and Object**

Classes have **fields** (that are ***private***) and **methods** (that are typically ***public***). There is one special method called a **Constructor**. The **constructor** is a method with the **same** **name** as the class. It is used to create (construct) an object. It is used with the ***new*** keyword.

The **fields** in an object are called **instance** variables. They are ‘module’ level variables not declared inside of methods. Using the key word “***this***” is a way to explicitly refer to the instance variables.

There are also special pairs of methods called ‘**get’** and ‘**set’** methods. These methods are used to set the value of a field and get the value of the field. They are the public interface to the value held inside the object. The advantage of using the **methods** is that we can do data validation as information is sent into the object and process the data as it leaves the object. We also use the methods to create **Read Only** variables (only the ‘**get’** method is public) or **Write Only** variables (one the ‘**set**’ method is public). The idea of encapsulation is possible when field variables are **private** and access to the field variables is controlled by **get/set** methods.

In some programming languages these are called “**Properties**” and they look like variables from outside the class.

**Creating Objects**

1. Create the variable

ClassName variableName;

1. Create the object with a **new** method AND
2. Assign the object to the variable in one statement

variableName = new ClassName (argument list);

Product product; //step 1

product = new Product (); //step 2 & 3

**Static versus instance (Object) methods and variables**

A **static** variable or method in a class is part of the class, not an object. To access the static element we use the **class** name, not the object name.

In the class the **static** keyword is used to define a variable or method as being part of the class.

public **static** int objectCount = 0;

Then to use the variable the **class** name is used in the code, not the object name.

countVariable = **Product**.objectCount ;