

Designing Classes

I. *Choosing Classes* (7.1)

Object-oriented programming is a major focus of AP Computer Science. The center of this programming paradigm in Java is a *class*. Both the AP CS A and AB Exams may test your ability to design and implement a class. (The AP CS AB Exam may also test your ability to decompose a problem into classes and to define the relationships and responsibilities of those classes.)

A class should represent a single concept from the problem description. Some examples of classes include: Rectangle, Coin, Purse, and Car. The properties of these classes were relatively easy to understand. The names of these classes are *nouns* that clearly identify the class. When you choose a class to implement, it should represent a single concept and should be named with a noun that easily identifies the class.

One particular category of classes is called an *actor* class. This class does work for you such as the Random class or StringTokenizer class. We also have *utility* classes, such as Math, that have no objects due to having all static methods. Note: The AP test does not have you distinguish between classes using the words actor or utility.

II. *Cohesion and Coupling* (7.2)

A class should represent one concept. All of the class responsibilities (interface features) should be closely related to the concept that the class represents. If they are, we say the class is *cohesive* or has a high degree of cohesion.

Some classes need other classes so that the class can do its job. A Purse class needs the Coin class because a purse contains coins. A roulette Game class needs the Spinner class to create and spin the roulette wheel. If many classes in a program depend on each other, we say that there is a high degree of *coupling*.

It is a good programming practice to have high cohesion and low coupling. If a high degree of coupling exists, then modifying one class may affect many other classes. (Figure 2 in Section 7.2 of your text illustrates high and low class coupling.) When designing a class, you should be able to list the class's responsibilities and collaborators (the classes that it uses or depends on). Doing so will help you see the amount of cohesion and coupling in your class design.

Let's look at an example of class responsibilities and collaborators. Consider the New York (or any state) lottery game. Each evening the lottery is televised and we observe the following. There is a *container* that holds the numbered *balls* such that no two balls have the same number. There is a *popper* that "pops" a random ball. It is this popped ball's number that is one of the lottery number choices.

Classes:

Container Class has a collection of Balls. If asked for a Ball, the Container will return the Ball that is requested.

- Responsibilities
 - hold Objects (Balls);
 - return an Object (Ball);
- Collaborators (other classes the Container needs or uses)
 - Ball

Ball Class has a number on it. It can return its number to whoever wants to know it.

- Responsibilities
 - return its number
- Collaborators (other classes the Ball needs or uses)
 - none

Popper Class will ask the Container how many Balls it has and then will ask the Random class for a random number generator to generate a random number in the range of the numbers on the Balls. The Popper will then ask the Container to remove a particular Ball.

- Responsibilities
 - ask Container for the number of Balls
 - ask Random for number
 - ask Container for Object (Ball) '
- Collaborators (other classes the Popper needs or uses)
 - Random
 - Container

Random Class (Java library class)

- Responsibilities
 - return a random number
- Collaborators (other classes Random needs or uses)
 - none

Is there any coupling in this example?

Answer: yes

III. *Accessor and Mutator Methods (7.3)*

Classes define the behavior of its objects by supplying methods. These methods either change the state of the object or they do not.

Methods that change the state of an object are called *mutator* methods or *modifiers* and usually have a void return type. Examples of modifiers include deposit and withdraw in the BankAccount class. Both of these methods change the state of the BankAccount object by modifying its balance. The methods drive and fillTank in the Car class change the state of a Car object by modifying its mileage and gasInTank.

Accessor methods do not change the state of the object. The method getBalance returns the balance of a BankAccount object without changing the state of the object. The method getMileage returns the total mileage of a Car object without changing the state of the object.

*A class with no mutator (modifier) methods is an immutable class.

Can you name an immutable class that we have used?

Answer: The String class is immutable. After a string has been constructed, its contents cannot change. The methods of the string class do not change the state of the String object.

Passing Parameters

In Java, a method can never change parameters of a primitive type. A method can change the state of an object reference but it cannot replace the reference with another. (See the example over the next three pages.)

```
public class Point
{
    public Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }
    public void setPoint(int xCoordinate, int yCoordinate)
    {
        x = xCoordinate;
        y = yCoordinate;
    }
    public int getX( )
    {
        return x;
    }
    public int getY( )
    {
        return y;
    }
    public String toString( )
    {
        String s = "(" + x + ", " + y + ")";
        return s;
    }
    public void sideEffect(Point second)
    {
        second = this;
        second.x = 20; // side effect - can be bad
        second.y = 10; // side effect - can be bad
    }
    private int x;
    private int y;
}
```

```
public class ParamsTest
{
    public static void main(String[ ] args)
    {
        // Trying to change primitive parameters
        int a = 10; int b = 11;
        System.out.println("Before call to change:");
        System.out.println("a = " + a + " b = " + b);
        change(a, b); // or ParamsTest.change (a, b); but this.change(a, b); not legal!
        System.out.println("After call to change: ");
        System.out.println("a = " + a + " b = " + b); // a = 10 b = 11
        System.out.println( );

        // Trying to change object references
        Point p1 = new Point(1, 2);
        Point p2 = new Point(3, 4);
        System.out.println("Before call to changePoints: ");
        System.out.println("p1 = " + p1); // p1 = (1, 2)
        System.out.println("p2 = " + p2); // p2 = (3, 4)
        changePoints(p1, p2); // or ParamsTest.changePoints (p1, p2);
        System.out.println("After call to changePoints: ");
        System.out.println("p1 = " + p1); // p1 = (1, 2)
        System.out.println("p2 = " + p2); // p2 = (3, 4)
        System.out.println( );

        // Changing the state of an object
        System.out.println("Before call to changeState:");
        System.out.println("p1 = " + p1); // p1= (1, 2)
        System.out.println("p2 = " + p2); // p2 = (3, 4)
        changeState(p1, p2); // or ParamsTest.changeState (p1, p2);
        System.out.println("After call to changeState:");
        System.out.println("p1 = " + p1); // p1 = (3, 4)
        System.out.println("p2 = " + p2); // p2 = (3, 4)
        System.out.println( );

        // Changing the state of an object by side effect code
        p1.sideEffect(p2);
        System.out.println("After call to sideEffect: ");
        System.out.println("p1 = " + p1); // p1 = (20, 10)
        System.out.println("p2 = " + p2); // p2 = (3, 4)
        System.out.println( );
    }
}
// more code on the next page
```

```
public static void change(int x1, int y1)
{
    x = x1;
    y = y1;
    int a = x1;
    int b = y2;
}

public static void changePoints(Point first, Point second)
{
    Point anotherPoint = new Point (9, 9);
    first = anotherPoint;
    second = first;
}

public static void changeState(Point first, Point second)
{
    first.setPoint(second.getX( ), second.getY( ));
}

int x;
int y;
}
```

Output of ParamsTest

Before call to change:

a = 10 b = 11

After call to change: a = 10 b = 11

Before call to changePoints:

p1 = (1, 2)

p2 = (3, 4)

After call to changePoints:

p1 = (1, 2)

p2 = (3, 4)

Before call to changeState:

p1 = (1, 2)

p2 = (3, 4)

After call to changeState:

p1 = (3, 4)

p2 = (3, 4)

After call to sideEffect:

p1 = (20, 10)

p2 = (3, 4)

This example shows that all parameters are copied into the parameter variables when a method starts. You may see this described as "primitive parameters are passed by value". With objects, the object *reference* is passed by value, not the object (i.e. the object state).

IV. *Side Effects* (7.4)

When a method changes the value of a variable other than its implicit parameter then that is called a side effect and can be a bad thing.

ex.1)

```
public void sideEffect (Point second)
{
    second = this;
    second.x = 20; // side effect - can be bad
    second.y = 10; // side effect - can be bad
}
```

V. *Preconditions and Postconditions* (7.5)

Preconditions and postconditions are statements that document the intended behavior of the method. They should appear in the method documentation. A *precondition* is a statement that describes the requirements that must be met in order for the method to complete its intended task correctly. If the precondition is not met, no promise about the method's behavior is made. It is the responsibility of the calling method to satisfy the precondition. If the precondition is not satisfied, how should the method behave? One way of handling a violation of the precondition is to throw an *exception* to indicate that the method was called inappropriately. Exceptions are introduced in Section 7.5 of our text and discussed more thoroughly in Chapter 14. (Throwing exceptions is included in the AB testable subset only.)

A *postcondition* is a promise that the value returned by the method is computed correctly or that the object is in a certain state. A postcondition's promise is valid only if the precondition is satisfied. The example on the next page contains information about a Product class. The class contains the name and the number of available products with this name. The method buyproduct shows a pre- and postcondition and throws an appropriate exception when the precondition is not met.

ex.1)

// Product contains information about products in a store

```
public class Product
{
    /* Constructs a product with given name and number available
       precondition: nm is the name of a product, number >= 0
    */
    public Product(String nm, int number)
    {
        numAvailable = number;
        name = nm;
    }

    //Postcondition: returns number of product available
    public int getNumAvailable( )
    {
        return numAvailable;
    }
    /*
       Precondition: n >= 0
       Postcondition: numAvailable is updated accounting for the
                       number of items bought
    */
    public void buyProduct (int numWanted)
    {
        if (numWanted > numAvailable)
            throw new IllegalArgumentException( );
        numAvailable -= numWanted;
    }
    // Other methods here
    private int numAvailable;
    private String name;
    // Other private stuff
}
```

pre- and postconditions

If you supply a method that only works correctly under certain conditions, be sure to use preconditions to document this fact. If you supply a method that is guaranteed to have a certain effect, use postconditions to document this fact. When you are using other classes in your programs, be sure to read the pre- and postconditions carefully. Remember that the calling method is responsible for satisfying the preconditions.

*Reading and understanding pre- and postconditions is important. Many times an algorithm for solving a free-response question on the AP test is given in the documentation.

VI. *Static Methods* (7.6)

Up to now, the methods we have included in our classes (except for Math methods) have been instance methods. These methods belong to the object and operate on the object that is instantiated. A static method belongs to the *class*, not to an object, and is sometimes referred to as a *class* method. A static method has no implicit parameter. Static methods are almost always invoked through a class and rarely, an object - i.e. `Class.methodName()`, not `obj.methodName()`. *A good programmer would call a static method by using the name of the class.

ex 1)

```
public class Class1
{
    public static void main (String[ ] args)
    {
        Class1 obj1 = new Class1( );
        this.method1( ); // compile error – NO “this” object in static main( )
        method1( ); // compile error – NO understood “this” in static main( )
        Class1.method1( ); // compile error – can never call a nonstatic
                           method with the name of the class
        obj1.method1( ); // method1( ) will be called
        obj1.method2( ); // method2( ) will be called but not a good programmer
        method2( ); // OK only because we are in a static method and going
                   to a static method in the same class
        Class1.method2( ); // method2( ) will be called
    }
    public void method1( )
    {
        System.out.println("method 1");
    }
    public static void method2( )
    {
        System.out.println ("method 2");
    }
}
```

VII. *Static Fields (7.7)*

The class variable, `numCarsMade`, is a static field (class field) of the `Car` class. The initialization of `numCarsMade` is done only once, when the `Car` constructor is called the first time. In the `Car` class shown on the next page, the class variable `numCarsMade` keeps a count of the number of `Car` objects that have been instantiated. Each time a `Car` object is created, `numCarsMade` is incremented by 1. The class method `getCarsMade` returns that number to the calling method.

ex.1)

```
System.out.println(Car.getCarsMade( ));
```

will print the value of `numCarsMade` which is the number of `Car` objects constructed.

ex.1)

```
public class Car
{
    // Default constructor initializes instance variables
    public Car( )
    {
        // Private instance variables initialized here
        numCarsMade++;    // Number of cars manufactured so far
    }

    // Other public methods here

    //return number of cars manufactured so far
    public static int getCarsMade( )
    {
        return numCarsMade; // or return Car.numCarsMade
    }

    private static int numCarsMade = 0;
}
```

Client code:

```
Car mustang = new Car( );
System.out.println(Car.getCarsMade( ));
Car charger = new Car( );
System.out.println(Car.getCarsMade( ));
```

The above will print the value of numCarsMade which is the number of Car objects constructed.

Note: mustang.numCarsMade, charger.numCarsMade, and Car.numCarsMade are all the same value. Any of these could be used in a method in the Car class but just numCarsMade is the “proper” one.

Note: Either mustang.getCarsMade() or charger.getCarsMade() could be used if the getCarsMade() method was not static and either one would then return the same answer. Car.numCarsMade() using a static method is the “proper” one and would return the same answer.

VII. *Scope* (7.8)

The word *scope* is used to describe the part of a program in which a variable is accessible. We have already discussed instance variables, local variables of methods, parameter variables, and variables declared in the initialization of **for** loops. The issue of variable scope is an important reason to choose variable names wisely. It is bad programming technique to give instance variables the same names as the parameter or local variables. If you do, the scope issues become complicated.

The scope of a local variable extends from the point of its declaration to the end of the block that encloses it. If a variable is declared in the loop initialization, it is accessible only within the loop. If a variable is declared at the start of a method, it is accessible in that method. Parameter variables in a method header are accessible only within that method. If you try to give two local variables with overlapping scopes the same name, the compiler will complain. You can have local variables with the same name if their scopes do not overlap. To avoid all of this confusion, use different names for different variables and never change the values of parameter variables.

Within a method of a class, you can access all other methods and all fields of that class by their simple names (without an object name prefix). If you are using a method outside the object, you must qualify it by prefixing the method name with the object name (for an instance method) or by the class name (for a class/static method). Whenever you see an instance method call without an implicit parameter (object name prefix), the method is called on the **this** parameter.

Initializing Variables:

The methods for initializing variables depend on the type of variable. The initialization of different variable types is summarized below:

- Local variables belong to an individual method. A local variable can be accessed only from within that method and must be initialized before you use it. Failure to initialize will cause the compiler to complain.
- Parameter variables also belong to an individual method. Parameter variables are initialized with the values that are supplied by the calling method.
- Instance fields belong to an object and static fields belong to a class and both can be used by all methods of its class. Instance fields should be initialized in the constructor of the class. If instance fields are not initialized explicitly, default values are assigned.
 - Numbers are initialized to zero
 - objects to null (including strings which are objects)
 - boolean values to false.

(Even though these default values are assigned to instance fields, it is good programming practice to initialize instance.)

ex. 1)

```
public double CarMpg(double miles, double gallons)
{
    return miles/gallons;
}

private double gallons;
private double miles;
```

Note: The miles and gallons used in the calculation miles/gallons have the values from the client program passed by the call.

What if in the CarMpg method you wanted to access the instance private fields gallons or miles? They are blocked and the local variables miles and gallons take precedence. It is said that the local variables *shadow* the instance fields.

How could you access the instance private fields gallons or miles from the CarMpg method?

Answer: use **this.gallons** or **this.miles**

ex. 2)

```
public class Example
{
    public static void main(String[ ] args)
    {
        Example ex1 = new Example( );
        Example ex2 = new Example( );
        int first = 100;
        System.out.println("main: " + ex1.doSomething(first));
        System.out.println("main: " + Example.doSomethingElse(first));
        System.out.println("main: " + Example.doSomethingAgain( ));
        System.out.println("main: " + ex1.doSomethingMore(first));
        System.out.println("main: " + ex2.doSomething(first));
    }
    public int doSomething(int first)
    {
        System.out.print("doSomething arrival: " + first + " ");
        this.first++;
        System.out.print("doSomething exit: " + first + " ");
        return this.first;
    }
    public int doSomethingMore(int first)
    {
        System.out.print("doSomethingMore arrival: " + first + " ");
        first++;
        System.out.print("doSomethingMore exit: " + first + " ");
        return first;
    }
    public static int doSomethingElse(int first)
    {
        System.out.print("doSomethingElse arrival: " + first + " ");
        first++;
        System.out.print("doSomethingElse exit: " + first + " ");
        return first;
    }
    public static int doSomethingAgain( )
    {
        System.out.print("doSomethingAgain arrival: " + first + " ");
        first++;
        System.out.print("doSomethingAgain exit: " + first + " ");
        return first;
    }
    private static int first = 200;
}
```

AP Programming - Chapter 7 Lecture

Output:

doSomething arrival: _____ doSomething exit: _____ main: _____

doSomethingElse arrival: _____ doSomethingElse exit: _____ main: _____

doSomethingAgain arrival: _____ doSomethingAgain exit: _____ main: _____

doSomethingMore arrival: _____ doSomethingMore exit: _____ main: _____

doSomething arrival: _____ doSomething exit: _____ main: _____

Output Answer:

doSomething arrival: 100 doSomething exit: 100 main: 201

doSomethingElse arrival: 100 doSomethingElse exit: 101 main: 101

doSomethingAgain arrival: 201 doSomethingAgain exit: 202 main: 202

doSomethingMore arrival: 100 doSomethingMore exit: 101 main: 101

doSomething arrival: 100 doSomething exit: 100 main: 203

VIII. Packages (7.9)

A package is a set of related classes. Several common packages that we use in Java are listed in Table 11 of Section 7.9 in your text. If you are using classes that belong to any of these packages (except java.lang), you must import the package. The import directive allows you to refer to the class of the package without fully qualifying the class name with the package prefix each time you use it.

ex.1)

```
import java.awt.Rectangle  
// Other stuff here  
Rectangle = new Rectangle(5, 10, 15, 20);
```

is much easier to read and understand than

```
java.awt.Rectangle = new java.awt.Rectangle(5, 10, 15, 20};
```

You are expected to have a basic understanding of packages and a reading knowledge of import statements. You will not be required to create packages on the AP Exam.

Also:

*When a variable or a method has package access, then all methods of classes in the same package can access that variable or method..

AP Programming - Chapter 7 Lecture

ex. 2)

```

public class Scope
{
    private int x = 0, c = 0;
    private static int b = 0;
    public static void main(String[] args)
    {
        int a;
        int b = 2;
        int x = 1;
        Scope ex = new Scope( );
        ex.mod1(x);
        a = mod2(b);
    }
    public void mod1(int c)
    {
        x = 2;
        c = c + x;
    }
    public static int mod2(int x)
    {
        x = 6;
        int c = 5;
        b = 1;
        return (x + b + c);
    }
}
    
```

Given the program to the left, fill in the table below with the appropriate abbreviations as described just below.

- 1) Under **Identifier Classification** write whether the variable in that method is:
L (local) or
E (an explicit passed parameter) or
I (an instance field variable) or
O (object reference)

- 2) Under **Can Be Referenced In Method** write:
Y (yes it is being used in that method) or
B (blocked from being used in that method) or
N (not accessible - can not be used in that method) or
T (could be used in that method if **this.** used) or
C (could be used in that method without **this.**)

<u>Variables</u>	<u>Identifier Classification</u>	<u>Can Be Referenced In Method:</u>		
		<u>main()</u>	<u>mod1()</u>	<u>mod2()</u>
<u>before main()</u>				
x	_____	___	___	___
b	_____	___	___	___
c	_____	___	___	___
<u>in main()</u>				
a	_____	<u>Y</u>	___	___
b	_____	<u>Y</u>	___	___
x	_____	<u>Y</u>	___	___
ex	_____	<u>Y</u>	___	___
<u>in mod1()</u>				
c	_____	___	<u>Y</u>	___
x	_____	___	<u>Y</u>	___
<u>in mod2()</u>				
x	_____	___	___	<u>Y</u>
b	_____	___	___	<u>Y</u>
c	_____	___	___	<u>Y</u>