JAVA PROGRAMMING
FOR THE
AP COMPUTER SCIENCE A EXAMINATION

FIRST EDITION

by

Leon Schram

John Paul II High School
Plano, Texas

SAMPLE PAGES

Note: The style of this book is to present a sequence of topics in a manner that places focus on any specific topic inside a grey table. The program examples, the program outputs and the topic-specific explanations are all located inside the table container, such as is used here.

General topics and important summaries are placed outside these topic tables.

Java Programming for AP® Computer Science A
Table of Contents

Java is a case-sensitive programming language.

Java is not an intelligent language. It is not a language that can determine what is meant when a minor spelling error is made. This goes even to the degree that identical words are not viewed to be the same when all characters are the same, except for case. When that happens, the language is said to be *case-sensitive*.

Print05.java

Program **`Print05`** will not compile. Human beings with intelligence can easily determine the intent of the program. Many programming languages are designed to be case-sensitive. Java is one such language. The **`System`** class starts with an upper-case **`S`**. The lower-case **`s`** of line 9 is not accepted. Likewise the upper-case **`P`** in **`Print`** is not allowed. The output does not show program output, but a compile error message.

```
 4 public class Print05
 5 {
 6   public static void main(String[ ] args)
 7   {
 8     System.out.print("THIS IS OK.");
 9     system.out.Print("THIS IS NOT OK.");
10   }
11 }
```

```
----jGRASP exec: javac -g Print05.java

Print05.java:9: error: package system does not exist
     system.out.Print("THIS IS NOT OK.");
            ^
1 error

----jGRASP wedge2: exit code for process is 1.
----jGRASP: operation complete.
```

Topic 2.1 Objects: Instances of Classes

At first glance it may appear that *classes* and *objects* are synonyms. What is a class? A class contains attributes and behaviors. What is an object? An object contains attributes and behaviors. There is a difference that will make sense soon. Computers have always imitated life. Spreadsheets did not suddenly appear as some nifty computer application. Around 1500 the Venetian merchants adopted a *double-entry bookkeeping* system. Scrolls from that period showed rows and columns that resembled a printed version of today's electronic spreadsheet. Incidentally, VisiCalc - the first spreadsheet software - was the first "killer" app or program for electronic computers.

In other words, objects are all around us. Do not go far. Check yourself out. You are an object with attributes and behaviors. You can also think of an object with nouns and verbs. You have eyes (attributes). Eyes see (behavior). You have a brain (noun) and the brain thinks (verb). Look at another object, such as a car. A car has a brake (noun) and the brake stops the car (verb).

Everything that was just mentioned applies to both classes and objects. So what is the difference? Think of class as a category and think of an object as one example or one instance of the category. A cat is a category and fluffy is one instance of a cat.

Classes	Objects
• A class has attributes. • A class has behaviors. • A class is a data type. • A class is a category. • Cat is a class. • Student is a class. • Teachers is a class.	• An objects has attributes. • A class has behaviors • An object is a variable. • An object is 1 instance or example of a class. • Whiskers is 1 instance of a cat. • Alexia is 1 instance of a student. • Mrs. Nelson is 1 instance of a teacher.

With primitive data types the creation of a program variable, and assigning its initial value, is easy. For instance, the program statement **`int age = 16;`** declares **`age`** to be an **`int`** variable with initial value of **`16`**. It is another story when classes and objects get into the picture. The class is the data type and the object is the variable. You are about to see some program examples that use a **`Bank`** class. You will rapidly learn that working with objects & classes is very different from working with primitive types.

Right now we are interested in doing what the title of the chapter states, which is **Using Objects**. Objects cannot be used until they are first created and the creation of an object requires the existence of a class. So let's take one example, a **`Bank`** class.

Bank.java

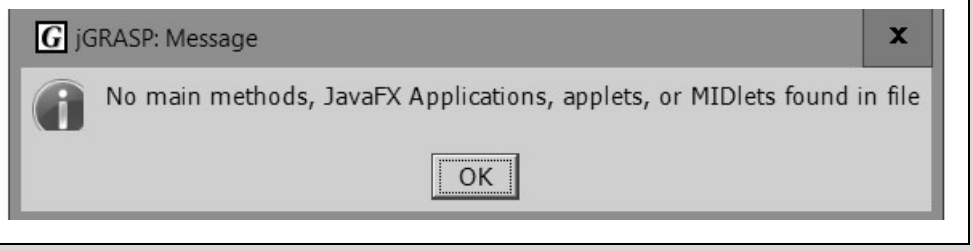
The **Bank.java** file contains the **Bank** class. It cannot execute by itself and it is used in this section to demonstrate concepts with the next two programs. The creation and requirements of classes will be handled in several future chapters.

```
 5 public class Bank
 6 {
 7   private double balance;
 8
 9   public Bank(double bal)
10   {
11     balance = bal;
12   }
13 }
```

The **Bank** class does compile. Classes are normally placed in their own files and it helps to know that a class is free of syntax errors. But that is all. Another testing program needs to determine if the **Bank** class functions properly. You can try to execute the **Bank** class, but if you do you will see the **jGRASP** message below stating that no **main** method is found.

```
----jGRASP exec: javac -g Bank.java

----jGRASP: operation complete.
```



Topic 5b.9 `this` Keyword

The word **`this`** is an interesting word. It has some special meanings that are surprisingly flexible. Consider the following situation between a student and a teacher. "Mr. Smith, may I borrow your laptop charger?" The teacher responds: "Yes, Kathy, but make sure to return the charger to me in *this* room." Where is "this" room. It is the room that the teacher occupies while talking to a student. If the same situation occurs in the cafeteria, then the students must return the charger to the teacher in the cafeteria.

How do we know where "this" is. It depends on the current context. If you are the student who asked the question then "this" is the room where you and the teacher are talking. In computer science, and in particular with Java, there is a special reference called **`this`**. It is an object and it behaves pretty much like the this word in English. Every person is located somewhere and every person can say: "Come to me tomorrow at noon at this location." Such a sentence is possible with millions of people.

The same applies to a Java program. The immediate or shallow value of an object is a reference to a logical location where attributes are stored. Every object that is properly constructed has its own specific reference value and every object can use the keyword **`this`** to indicate their own specific reference value. So how does one know which **`this`** and which value is appropriate? It is similar to human situations. You are standing in a room while talking. The room at that moment is the location when somebody says: "come back to *this* room." In a computer program of thousands of lines, one line at a time is being executed and in the Object Oriented environment of your Java program, you are in some container. You probably arrived at some location, because of a method call with some object parameter. The `this` reference will be the same value as the object reference that is currently being used. So far you have enough information to be confused. There have been some examples of using **`this`** in previous programs. Talking about scope involves using **`this`**. In the current section you will see a sequence of programs to make more sense of the mysterious reference **`this`**.

The this Reference

`this` is a reference. It stores the exact same value that references a memory location as an object. **`this`** can be many difference values, which depend on the current object in context of the program sequence.

The first program example, **`This01`**, constructs two **`Widget`** objects. The program then proceeds to print the values of the two objects. There is also a **`print`** statement inside the constructor where the value of the **`this`** reference is printed.

This01.java

Line 8 calls the constructor. The first output is from the constructor, line 22. This is followed by printing the value of **`this`** in line 23. Then the program sequence returns to line 10 and displays the value of **`widget1`**. The **`toString`** method is not defined in the **`Widget`** class, it will display the reference value. Note that **`this`** and **`widget1`** are the same value. The same process is repeated with the second object, **`widget2`** and **`this`** is now the same as **`widget2`**.

```
 4 public class This01
 5 {
 6   public static void main(String[ ] args)
 7   {
 8     Widget widget1 = new Widget();
 9     System.out.println("Inside main method");
10     System.out.println(widget1);
11     Widget widget2 = new Widget();
12     System.out.println("Inside main method");
13     System.out.println(widget2);
14   }
15 }
16
17 class Widget
18 {
19   public Widget()
20   {
21     System.out.println("Inside Widget constructor");
22     System.out.println(this);
23   }
24 }
```

```
----jGRASP exec: java This01

 Inside Widget constructor
 Widget@15db9742
 Inside main method
 Widget@15db9742
 Inside Widget constructor
 Widget@6d06d69c
 Inside main method
 Widget@6d06d69c
```

The `selectionSort` method with a static array

The **`selectionSort`** method shows the familiar three statements that swap values. Is it not the intent of the improved *Selection Sort* to reduce swapping? Yes, but the intent is to reduce, not eliminate. In a list of **`10,000`** numbers the *Bubble Sort* swap numbers as many as **`9,999`** times, while the *Selection Sort* only swaps once. This is a considerable improvement.

Sorting02a.java

The **`selectionSort`** method shares the outer-loop concept of the **`bubbleSort`**. It repeats comparison passes. The algorithmic logic of the sort routine is shown with the inner loop.

In line 18 the smallest value is stored, or technically, the index of the smallest value is stored. Lines 19, 20 and 21 compare all the array elements with **`smallest`** and reassign **`smallest`** any time a smaller value is found.

Only after the entire inner-loop is finished do you actually see the three swap statements of lines 22, 23 and 24 jump into action. That is right, the swap routine is part of the outer loop, not the inner loop.

```
14   public static void selectionSort (int[ ] list)
15   {
16     for (int p = 0; p < list.length; p++)
17     {
18       int smallest = p;
19       for (int q = p+1; q < list.length; q++)
20         if (list[q] < list[smallest])
21           smallest = q;
22       int temp = list[p];
23       list[p] = list[smallest];
24       list[smallest] = temp;
25     }
26   }
```

```
----jGRASP exec: java Sorting02a

23 89 55 17 99 45 11 60 67 34 81 28 31 15 72
11 15 17 23 28 31 34 45 55 60 67 72 81 89 99
```

The `selectionSort` method with a dynamic array

There is a very important goal intended for Chapter 7. It is not simply to introduce the `ArrayList` class and demonstrate various algorithms. An important goal is to gain comfort with coding using both the static array and the **`ArrayList`** class. Comfort means you can easily write program code in either one of the arrays. For this reason, much has been shown in both static and dynamic code.

Sorting02b.java

The **`selectionSort`** compares every **`list`** element, but the comparison is with a **`list`** element and the current smallest **`list`** element. This program has also made a change by placing the three swap program statements in its own **`swap`** method. Think question! Does our **`selectionSort`** method benefit with faster execution time by using a `swap` method?

```
19  public static void selectionSort (ArrayList<Integer> list)
20  {
21    for (int p = 0; p < list.size(); p++)
22    {
23      int smallest = p;
24      for (int q = p+1; q < list.size(); q++)
25        if (list.get(q) < list.get(smallest))
26          smallest = q;
27      swap(list,p,smallest);
28    }
29  }
30
31  private static void swap (ArrayList<Integer> list, int x, int y)
32  {
33    int temp = list.get(x);
34    list.set(x,list.get(y));
35    list.set(y,temp);
36  }
```

```
----jGRASP exec: java Sorting02b

[23, 89, 55, 17, 99, 45, 11, 60, 67, 34, 81, 28, 31, 15, 72]
[11, 15, 17, 23, 28, 31, 34, 45, 55, 60, 67, 72, 81, 89, 99]
```

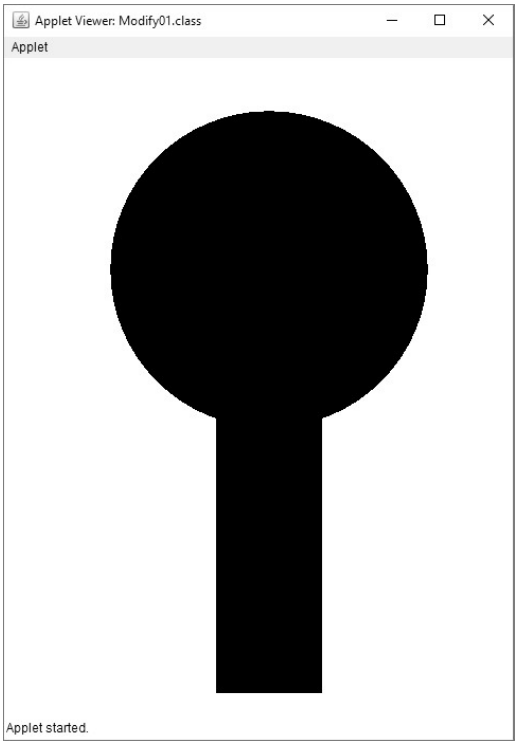
Topic 9.3 Overriding Methods

Graphics is not part of the APCS A Framework. but visual graphics is well suited for showing subclasses that override or redefine methods. It is also possible to do nothing different in the subclass. Perhaps that seems strange, but remember reliability. The creation with a subclass may be testing that everything works correctly, like its superclass, and then proceed to make changes.

Modify01.java

Program **`Modify01`** does not modify anything and does not demonstrate any inheritance. It is a program that draws a simple tree. It will become the superclass for the future inheritance program examples.

```
 8 public class Modify01 extends Applet
 9 {
10    public void paint(Graphics g)
11    {
12       Tree tree = new Tree();
13       tree.drawTree(g);
14    }
15 }
16
17 class Tree
18 {
19    public void drawTree(Graphics g)
20    {
21       g.setColor(Color.black);
22       drawTrunk(g);
23       drawLeaves(g);
24    }
25
26    public void drawTrunk(Graphics g)
27    {
28       g.fillRect(200,300,100,300);
29    }
30
31    public void drawLeaves(Graphics g)
32    {
33       g.fillOval(100,50,300,300);
34    }
35 }
```

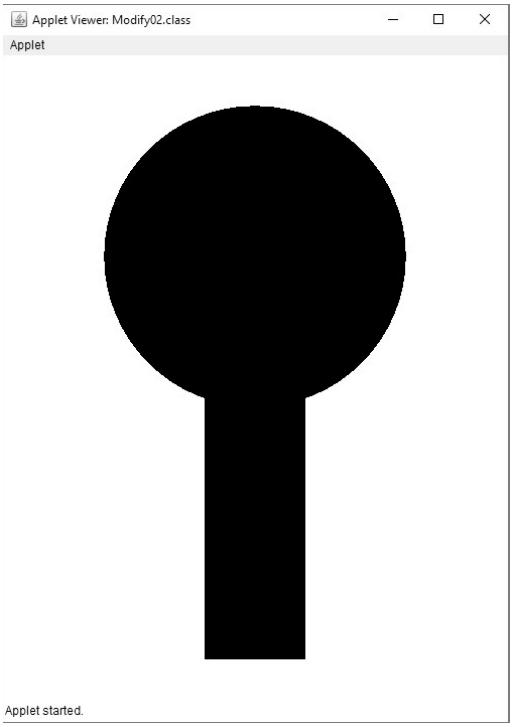


A subclass can change nothing at all.

Modify02.java

The **SubTree** class in lines 36..38 shows an empty class body. It will inherit all the capabilities from the **Tree** class and change nothing. The tree displayed by the **SubTree** object is identical in appearance to the superclass **Tree**.

```
 7 public class Modify02 extends Applet
 8 {
 9   public void paint(Graphics g)
10   {
11     SubTree subTree = new SubTree();
12     subTree.drawTree(g);
13   }
14 }
15
16 class Tree
17 {
18   public void drawTree(Graphics g)
19   {
20     g.setColor(Color.black);
21     drawTrunk(g);
22     drawLeaves(g);
23   }
24
25   public void drawTrunk(Graphics g)
26   {
27     g.fillRect(200,300,100,300);
28   }
29
30   public void drawLeaves(Graphics g)
31   {
32     g.fillOval(100,50,300,300);
33   }
34 }
35
36 class SubTree extends Tree
37 {
38 }
```

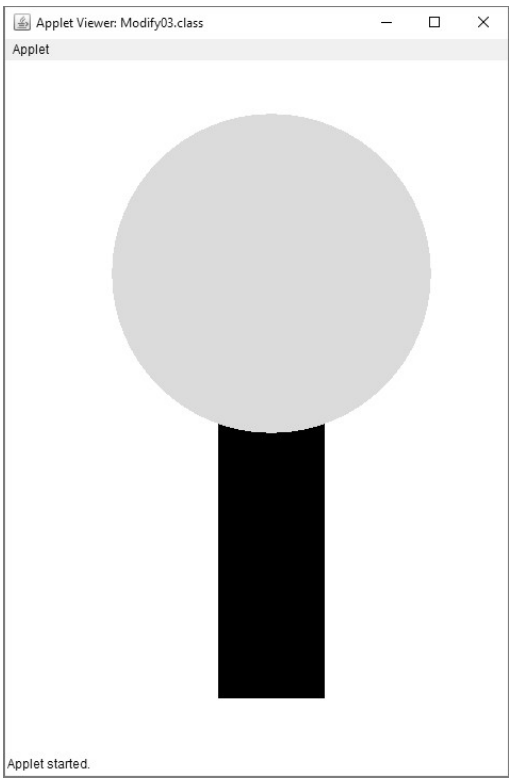


A subclass can redefine one or more methods.

Modify03.java

The **SubTree** class in program **Modify03** redefines method **drawLeaves**. The leaves are now green. In a printed version the leaves will appear to be a shade of gray.

```
 8 public class Modify03 extends Applet
 9 {
10   public void paint(Graphics g)
11   {
12     SubTree subTree = new SubTree();
13     subTree.drawTree(g);
14   }
15 }
16
17 class Tree
18 {
19   public void drawTree(Graphics g)
20   {
21     g.setColor(Color.black);
22     drawTrunk(g);
23     drawLeaves(g);
24   }
25
26   public void drawTrunk(Graphics g)
27   {
28     g.fillRect(200,300,100,300);
29   }
30
31   public void drawLeaves(Graphics g)
32   {
33     g.fillOval(100,50,300,300);
34   }
35 }
36
37 class SubTree extends Tree
38 {
39   public void drawLeaves(Graphics g)
40   {
41     g.setColor(Color.green);
42     g.fillOval(100,50,300,300);
43   }
44 }
```

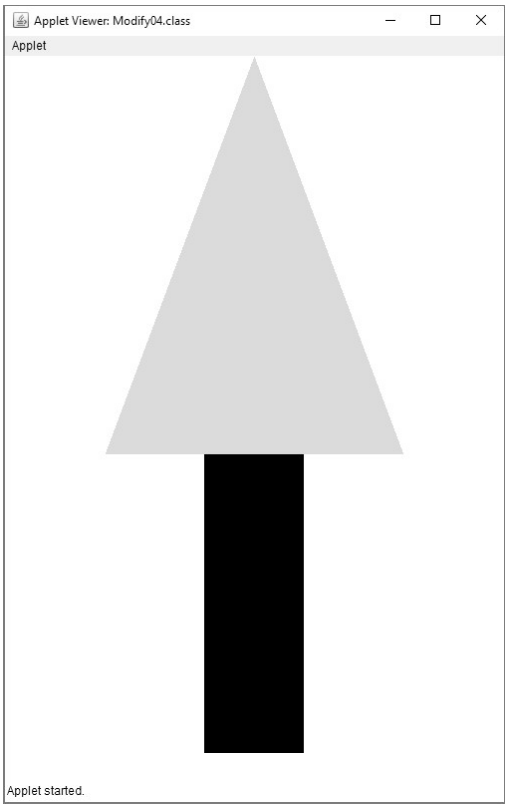


The `PineTree` subclass overrides method `drawLeaves`.

Modify04.java

The name of the subclass is now **`PineTree`**. It is good to use self-documenting identifiers whenever possible. This time method **`drawLeaves`** changes both the color and the shape of the superclass **`Tree`** appearance.

```
12      PineTree pineTree = new PineTree();
13      pineTree.drawTree(g);
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
16
17 class Tree
18 {
19    public void drawTree(Graphics g)
20    {
21       g.setColor(Color.black);
22       drawTrunk(g);
23       drawLeaves(g);
24    }
25
26    public void drawTrunk(Graphics g)
27    {
28       g.fillRect(200,400,100,300);
29    }
30
31    public void drawLeaves(Graphics g)
32    {
33       g.fillOval(100,150,300,300);
34    }
35 }
37
38 class PineTree extends Tree
39 {
40    public void drawLeaves(Graphics g)
41    {
42       g.setColor(Color.green);
43       int x = 200;
44       int y = 400;
45       Polygon triangle = new Polygon();
46       triangle.addPoint(x+50,y-400);
47       triangle.addPoint(x+200,y);
48       triangle.addPoint(x-100,y);
49       g.fillPolygon(triangle);
50    }
51 }
```



A subclass can newly-define one or more methods.

Modify05.java

Program **`Modify05`** creates a new subclass **`XmasTree`**, which is a subclass of **`PineTree`**. It inherits various methods from **`Tree`** and **`PineTree`** and then newly-defines **`drawOrnaments`**. Please notice that at every level as few changes as possible are made, because of Inheritance.

```
12      XmasTree xmasTree = new XmasTree();
13      xmasTree.drawTree(g);
14      xmasTree.drawOrnaments(g);
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
17
18 class Tree
19 {
20    public void drawTree(Graphics g)
21    // Same code as previous program
26
27    public void drawTrunk(Graphics g)
28    // Same code as previous program
31
32    public void drawLeaves(Graphics g)
33    // Same code as previous program
36 }
37
38 class PineTree extends Tree
39 // Same code as previous program
52
53 class XmasTree extends PineTree
54 {
55    public void drawOrnaments(Graphics g)
56    {
57       int x = 200;
58       int y = 500;
59       g.setColor(Color.red);
60       g.fillOval(x-50,y-60,30,30);
61       g.fillOval(x+40,y-60,30,30);
62       g.fillOval(x+130,y-60,30,30);
63       g.fillOval(x,y-140,30,30);
64       g.fillOval(x+90,y-140,30,30);
65       g.fillOval(x+50,y-200,30,30);
66       g.fillOval(x+20,y-260,30,30);
67       g.fillOval(x+50,y-200,30,30);
68       g.fillOval(x+40,y-320,30,30);
69    }
70 }
```