Herbert Schildt's *Java: A Beginner's Guide*, 6th ed.

Chapter 6 Self Test Answers

1. **Given two fragments below, is the second one correct?**

   Fragment 1:

```
class X {
  private int count;
}
```

   Fragment 2:

```
class Y {
  public static void main(String args[]){
    X ob = new X();

    ob.count = 10;
  }
}
```

   This code fragment is not correct. The *count* variable has an access modifier of *private* and therefore it is not possible to access this variable directly outside of its class. The code requires a getter method such as the one below, which controls access to the variable outside the class:

```
class X {
  private int count;

  public int getCount(){
    return count;
  }
}
```

2. **An access modifier must _____ a member's declaration.**

   An access modifier must *precede* a member's declaration.

3. **The complement of a queue is a stack. It uses first-in, last-out accessing and is often likened to a stack of plates. The first plate put on a table is the last plate used. Create a stack class called *Stack* that can hold characters. Call the methods that access the stack *push()* and *pop()*. Allow the user to specify the size of the stack when it is created. Keep all other members of the *Stack* class private. (Hint: You can use the *Queue* class as a model; just change the way the data is accessed.)**

   The following code was copied directly from the answer in the back of the book as I was unsuccessful in creating a Stack class on my own:

```
class Stack {
  private char stack[]; // this array holds the stack
  private int tos; // top of stack

  // Construct an empty Stack given its size
  Stack(int size){
    stack = new char[size]; // allocate memory for stack
    tos = 0;
  }

  // Construct a Stack from a Stack
  Stack (Stack ob){
    tos = ob.tos;
    stack = new char [ob.stack.length];

    // copy elements
    for (int i=0; i < tos; i++){
      stack[i] = ob.stack[i];
    }
  }

  // Construct a stack with initial values
  Stack(char a[]){
    stack = new char[a.length];

    for (int i = 0; i < a.length; i++){
      push(a[i]);
```

```
    }
  }

  // Push characters onto the stack
  void push(char ch){
    if (tos == stack.length){
      System.out.println(" -- Stack is full.");
      return;
    }

    stack[tos] = ch;
    tos++;
  }

  // Pop a character from the stack
  char pop(){
    if (tos == 0){
      System.out.println(" -- Stack is empty.");
      return (char) 0;
    }

    tos--;
    return stack[tos];
  }

} // end Stack

// Demonstrate the Stack Class
public class StackDemo {
  public static void main(String args[]){
      // construct a 10-element empty stack
      Stack stack1 = new Stack(10);

      char name[] = {'B', 'r', 'a', 'd'};

      // construct stack from array
      Stack stack2 = new Stack(name);

      char ch;
      int i;

      // put some characters into stack1
      for (i=0; i < 10; i++){
        stack1.push((char)('1' + i));
      }

      // construct stack from another stack
      Stack stack3 = new Stack(stack1);

      System.out.println();

      // show the stacks
      System.out.printf("Contents of stack1: ");
      for (i=0; i < 10; i++){
        ch = stack1.pop();
        System.out.print(ch);
      }

      System.out.println("\n");

      System.out.printf("Contents of stack2: ");
      for (i=0; i< 4; i++){
        ch = stack2.pop();
        System.out.print(ch);
      }

      System.out.println("\n");

      System.out.printf("Contents of stack3: ");
      for(i=0; i < 10; i++){
        ch = stack3.pop();
        System.out.print(ch);
```

```
        }

        System.out.println("\n");
    } // end main
} // end StackDemo
```

4. **Given the class below, write a method called *swap()* that exchanges the contents of the objects referred to by two *Test* reference objects.**

```
class Test{
  int a;
  Test(int i){
    a = i;
  }
}
```

The following class satisfies the specifications requested above. I chose to use a static method for swap, because it seemed that there only needed to be one swap method per class, there is no need for each individual object to have its own swap method.

```
class Test{

  int a = 0;

  Test(int i){
    a = i;
  }

  // Swap contents of 2 test objects
  static void swap(Test x, Test y){
    Test temp = new Test(0);
    Test temp01 = x;
    Test temp02 = y;
    temp.a = temp01.a;
    temp01.a = temp02.a;
    temp02.a = temp.a;
  }

  public static void main(String[] args){

    Test test1 = new Test(1);
    Test test2 = new Test(2);

    System.out.println();
    System.out.println("Value of test1: " + test1.a);
    System.out.println("Value of test2: " + test2.a + "\n");

    Test.swap(test1, test2);

    System.out.println("Value of test1: " + test1.a);
    System.out.println("Value of test2: " + test2.a + "\n");
  } // end main
} // end Test
```

5. **Is the following fragment correct?**

```
class X {
  int meth(int a, int b) { ... }
  String meth(int a, int b) { ... }
}
```

This fragment seems to be attempting to overload the same *meth()* method. While it is possible to have two different return types (in this case *int* and *String*), it is not possible for two overloaded methods to have the same parameter signature. Therefore, this code fragment would not compile.

6. **Write a recursive method that displays the contents of a string backwards.**

   The following program utilizes recursion to display the contents of a string backwards:

```
class RecursionDemo {

  String stringToReverse;

  RecursionDemo(String x){
      stringToReverse = x;
  }

  public void reverse(int index){
    if (index != stringToReverse.length() -1){
      reverse (index+1);
    }
    System.out.print(stringToReverse.charAt(index));
  }

  public static void main (String[] args){

    RecursionDemo demonstrateRecursion = new RecursionDemo("Java is Spectacular!!"

    System.out.println("\n" + "The original statement is: " + demonstrateRecursion
    System.out.printf("After being reversed via recurision it becomes: ");
    demonstrateRecursion.reverse(0);
    System.out.println("\n");

  } // end main
} // end Recursion
```

7. **If all objects of a class need to share the same variable, how must you declare that variable?**

   If all the objects of a class need to share the same variable, it must be declared as *static*, which makes one copy of the variable per class, not per object. A common example of this procedure would be to declare a *public static int counter = 0;* variable, and each time a new object is created, the *constructor* would increment the static counter variable. There are many examples such as this of the uses of *static* variables.

8. **Why might you need to use a *static* block?**

   A *static block* is executed when the class is first loaded, and is therefore useful if a class requires some type of initialization before it is ready to create objects (for example, establish a connection to a remote site). Another possibility is the initialization of static variables before the creation of objects.

9. **What is an inner class?**

   An inner class is a specialized type of *non-static nested class*. An inner class appears inside an outer class and provides a set of services that is only used by the outer class. It has access to all of the variables and methods of the outer class and refer to them directly in the same way that other non-static members of the outer class do. An example of an inner class would be as follows:

```
class Outer {

  String example = "This is some sample text.";

  class Inner {

    System.out.println("The inner class is printing: " + example);

  } // end Inner

} // end Outer
```

10. **To make a member accessible by only other members of its class, what access modifier must be used?**

    In Java, there are three (3) access modifiers that can be placed in from of class members, *public*, *private*, and *protected*. *public*, predictably, makes the member accessible by any member of any class. *private* makes a variable only be accessible by other members of its class. *protected* means that the member is accessible from within its own class and from within its *package*.

11. **The name of a method plus its parameter list constitutes the method's _____.**

    The name of a method plus its parameter list constitutes the method's *signature*.

12. **An *int* argument is passed to a method by using the call-by-_____.**

    An *int* argument is passed to a method by using the *call-by-value*. This means that the *value* of the member into the formal parameter of the subroutine. Therefore, changes made to the parameter by the subroutine have no effect on the member. Objects, by contrast, use *call-by-reference*, which means a refernce to the object itself is passed as a parameter, so any changes made by the method are changes that are made to the object itself.

13. **Create a varargs method called *sum()* that sums the *int* values passed to it. Have it return the result. Demonstrate its use.**

    The following program satisfies the criteria outlined above:

```
class VarargsDemo{

  static int sumOfInts = 0;

  public static int sum(int ... v){
    for (int i =0; i < v.length; i++){
      sumOfInts += v[i];
    }
    return sumOfInts;
  }

  public static void main(String[] args){

    sum(53, 97, 412);

    System.out.println();
    System.out.println("The sum of the integers is: " + sumOfInts + "\n");

  } // end main
} // end VarargsDemo
```

14. **Can a varargs method be overloaded?**

    Varargs methods can be overloaded, but with caution as an overloaded varargs method could cause ambiguity. The types of varargs parameter can differ or other (normal, non-varargs parameters) could be added to the method. In many cases, it is easier to simply use a variety of names for the methods instead of overloading.

15. **Show an example of an overlooked varargs method that is ambiguous.**

    If the following methods were created:

```
static void example (int ... v){};

static void example (int i, int ... v){};
```

    there would be no way for the compiler to tell which of the two methods was being called if it were passed, for example:

```
vaTest(1);
```

    Therefore, this code would generate an error and not be correct. It is much more simple to create multiple methods with multiple names than to overload a method that could create ambiguity for the compiler.