Answers to review questions from Chapter 2

1. Explain in your own words the difference between a *method* and a *program*.

   A *method* computes a value or performs some operation on behalf of the code for a program. A *program* executes computation on behalf of a user. A program typically includes many methods.

2. Define the following terms as they apply to methods: *call*, *argument*, *return*.

   In programming terminology, the process of invoking a method is referred to as *calling* that method. In the process of making a call, the caller can provide data to the method in the form of *arguments,* which are a set of local variables that are initialized from the values written inside the parentheses that designate the call. When the method completes its work, it *returns* to its caller, often passing back a value as a result.

3. Can there be more than one `return` statement in the body of a method?

   Yes. A method can include any number of `return` statements.

4. What is a *static method?*

   A *static method* is a method associated with a class rather than with a specific object. Static methods are called by including the class name and a dot before the method name.

5. How would you calculate the trigonometric sine of 45° using the `Math` class?

   ```
   Math.sin(Math.toRadians(45))
   ```

6. What is a *predicate method?*

   A *predicate method* is a method that returns a Boolean value.

7. Describe the difference in role between an *implementer* and a *client.*

   The *implementer* is responsible for writing the actual code that carries out an operation. The *client* invokes that code to perform the operation but need not understand the details for doing so.

8. What is meant by the term *overloading?* How does the Java compiler use *signatures* to implement overloading?

   In Java, the term *overloading* refers to the fact that you can define several different methods with the same name, as long as each method has a different *signature,* which indicates the number and types of the arguments. When the Java compiler sees a call to an overloaded method with a particular name, it examines the argument values to see which version of that method is required.

9. What is a *stack frame?*

   A *stack frame* is the region of memory used to hold the values of local variables during a method call. The frame is created when the method is called and deleted when the method returns.

10. Describe the process by which *arguments* are associated with *parameters.*

    Arguments and parameters are matched by their order in the argument list. The names of the parameters have no bearing on the association process.

11. Variables declared within a method are said to be *local variables*. What is the significance of the word *local* in this context?

    Local variables can be used only within that method. Neither the caller nor any method called from inside a method has access to those local variables.

12. What is the difference between *iteration* and *recursion?*

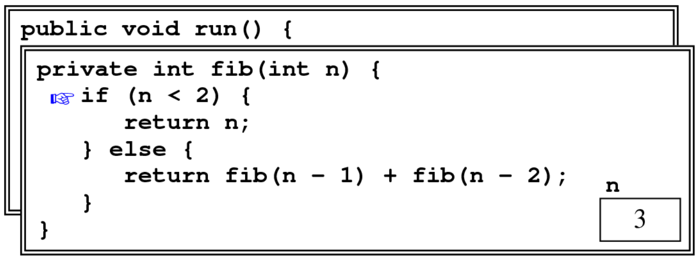
    Iteration refers to the process of performing repeated calculations by using a loop to execute the same code.

13. What is meant by the phrase *recursive leap of faith?* Why is this concept important to you as a programmer?

    The *recursive leap of faith* refers to the idea that, in writing a recursive solution, you can assume that smaller instances of the problem work and then reassemble those solutions to solve the original problem. If you fail to adopt the recursive leap of faith, you are forced to trace the operation of the program all the way down to the simple cases, which often makes the process too complex to follow.

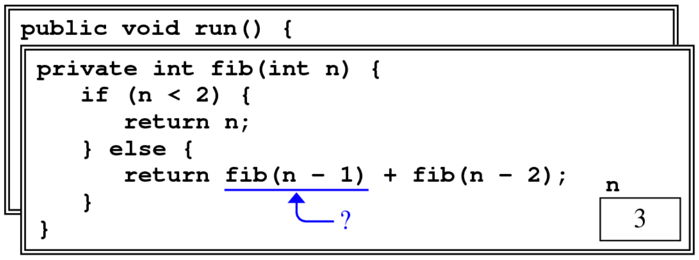
14. In the section entitled "Tracing the recursive process," the text goes through a long analysis of what happens internally when `fact(4)` is called. Using this section as a model, trace the execution of `fib(3)`, sketching out each stack frame created in the process.

Step 1:



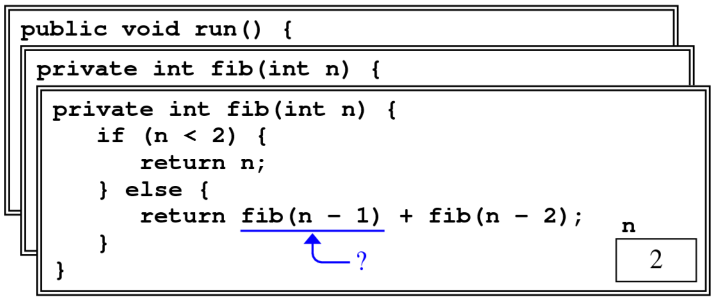
```
public void run() {
  private int fib(int n) {
     if (n < 2) {
        return n;
     } else {
        return fib(n - 1) + fib(n - 2);
     }
  }
```

Step 2:



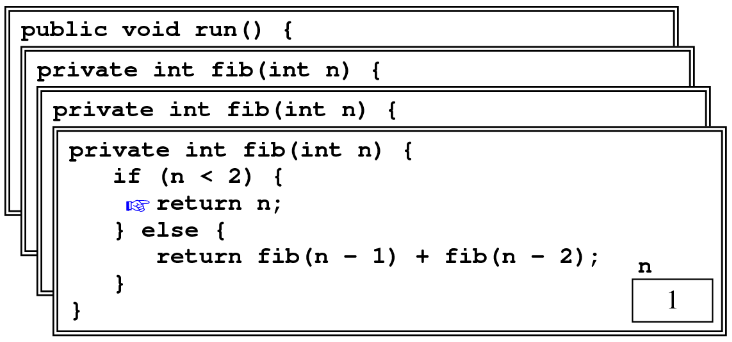
```
public void run() {
  private int fib(int n) {
     if (n < 2) {
        return n;
     } else {
        return fib(n - 1) + fib(n - 2);
     }
  }
```

Step 3:



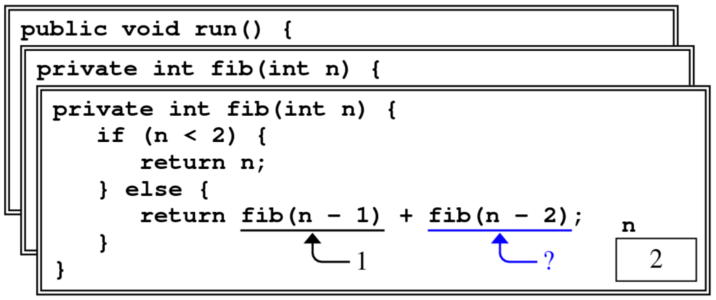
```
public void run() {
  private int fib(int n) {
    private int fib(int n) {
       if (n < 2) {
          return n;
       } else {
          return fib(n - 1) + fib(n - 2);
       }
    }
```

Step 4:



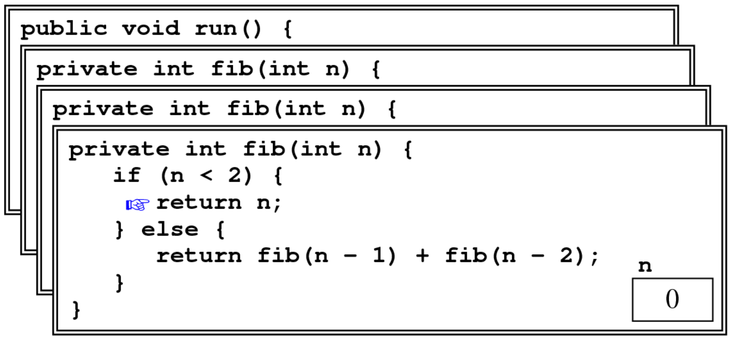
```
public void run() {
 private int fib(int n) {
  private int fib(int n) {
   private int fib(int n) {
      if (n < 2) {
       ☞ return n;
      } else {
         return fib(n - 1) + fib(n - 2);
      }
   }
```

Step 5:



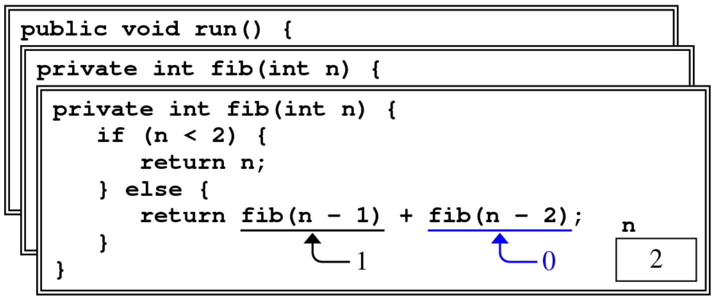
```
public void run() {
 private int fib(int n) {
  private int fib(int n) {
     if (n < 2) {
        return n;
     } else {
        return fib(n - 1) + fib(n - 2);
     }
  }
```

Step 6



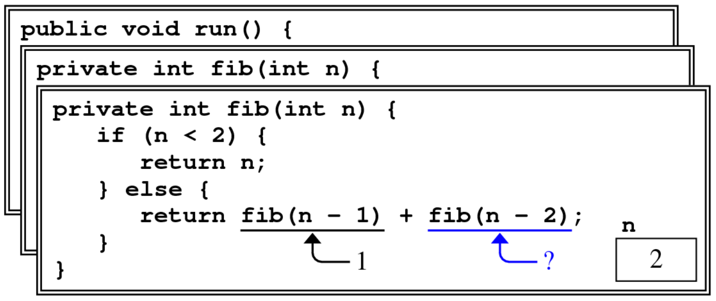
```
public void run() {
 private int fib(int n) {
  private int fib(int n) {
   private int fib(int n) {
      if (n < 2) {
       ☞ return n;
      } else {
         return fib(n - 1) + fib(n - 2);
      }
   }
```

Step 7:



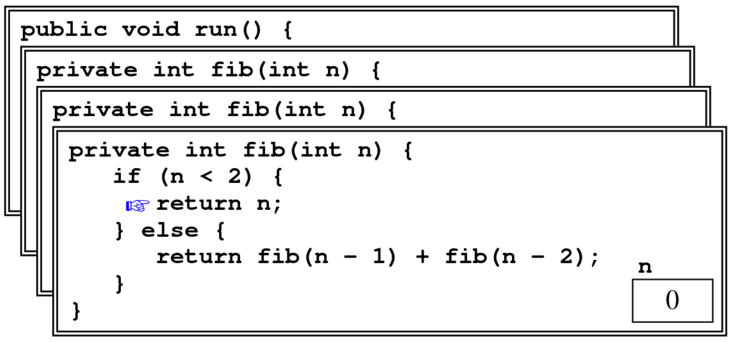
```
public void run() {
 private int fib(int n) {
  private int fib(int n) {
     if (n < 2) {
        return n;
     } else {
        return fib(n - 1) + fib(n - 2);
     }
  }
```

Step 8:



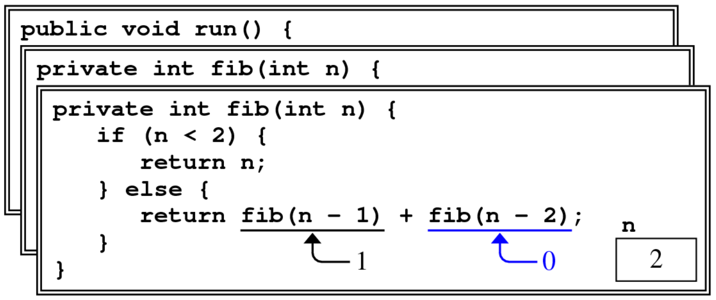
```
public void run() {
 private int fib(int n) {
  private int fib(int n) {
     if (n < 2) {
        return n;
     } else {
        return fib(n - 1) + fib(n - 2);
     }
  }
```

Step 9:



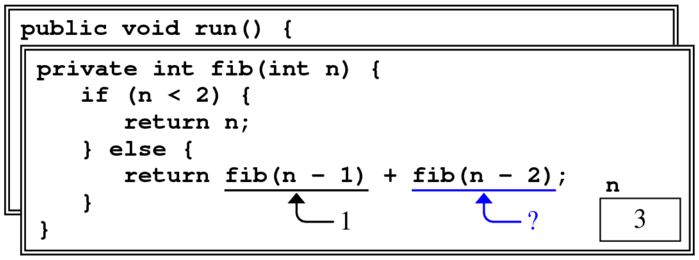
```
public void run() {
private int fib(int n) {
private int fib(int n) {
private int fib(int n) {
   if (n < 2) {
   ☞ return n;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

Step 10:



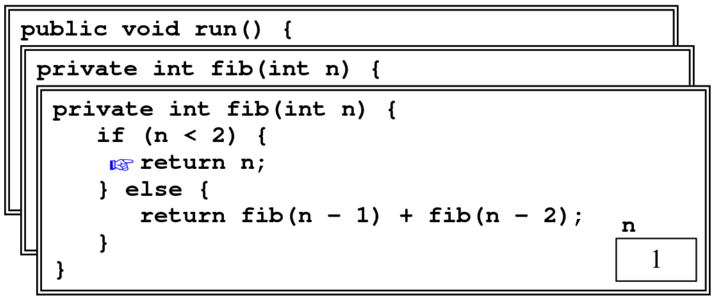
```
public void run() {
private int fib(int n) {
private int fib(int n) {
   if (n < 2) {
      return n;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

Step 11:



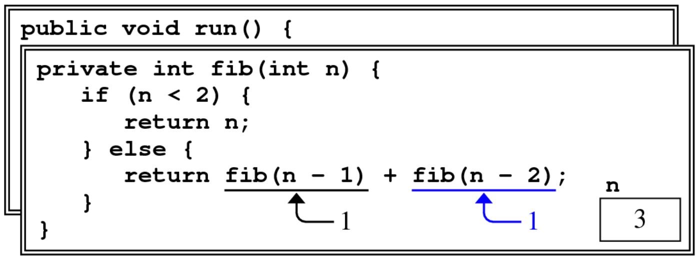
```
public void run() {
private int fib(int n) {
   if (n < 2) {
      return n;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

Step 12:



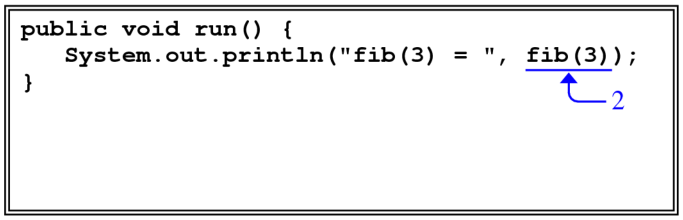
```
public void run() {
private int fib(int n) {
private int fib(int n) {
   if (n < 2) {
   ☞ return n;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

Step 13:



```
public void run() {
private int fib(int n) {
   if (n < 2) {
      return n;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

Step 14:



```
public void run() {
   System.out.println("fib(3) = ", fib(3));
}
```

15. What is a *recurrence relation?*

    A *recurrence relation* is an expression defining each new element of a sequence in terms of earlier elements.

16. Modify Fibonacci's rabbit problem by introducing the additional rule that rabbit pairs stop reproducing after giving birth to *three* litters. How does this assumption change the recurrence relation? What changes do you need to make in the simple cases?

    The new recurrence relation must subtract out the rabbits that are too old to reproduce, which are all of those born more than four months ago. The recurrence relation therefore looks like this:

    $$t_n = t_{n-1} + t_{n-2} - t_{n-5}$$

    The simple case must now take account of the fact that the value of *n* may now range into negative territory, when there were no rabbits. The code for the revised rabbit problem looks like this:

```
private int rabbits(int n) {
   if (n <= 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   } else {
      return rabbits(n - 1) + rabbits(n - 2) - rabbits(n - 5);
   }
}
```

17. How many times is `fib(1)` called when `fib(n)` is calculated using the recursive implementation given in Figure 2-5?

    The method `fib(1)` is called `fib(n)` times.

18. What is a *wrapper method?* Why are wrapper methods often useful in writing recursive methods?

    A *wrapper method* is a method whose only role is to set things up for a call to a more general recursive method that does all the work. In many situations, the recursive process requires additional state information that the wrapper method can provide.

19. What would happen if you eliminated the `if (n == 1)` check from the method `additiveSequence`, so that the implementation looked like this:

```
int additiveSequence(int n, int t0, int t1) {
   if (n == 0) return t0;
   return additiveSequence(n - 1, t1, t0 + t1);
}
```

    Would the method still work? Why, or why not?

    The method would still work. In the absence of the check, the method calls itself recursively with 0 as the first argument and the value of `t1` as the second. Since `n` will then be 0 in the recursive call, the method will return the original value of `t1`, just as before.