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

Chapter 2 Self Test Answers

1. **Why does Java strictly specify the range and behavior of its primitive types?**

   Schildt states on page 33 that "Java strictly specifies a range an behavior for each primitive type, which all implementations of the Java Virtual Machine must support. Because of Java's portability requirement, Java is uncompromising in this account." Clearly because Java needs to be cross-platform (write-once, read anywhere) it has to be very strict about how to properly use its variables.

2. **What is Java's character type, and how does it differ from the character type used by some other programming languages?**

   In Java, a *char* variable is a Unicode value, which can represent all the characters found in human language. A *char* can be represented as an integer or as a letter (surrounded by single quotes) as so:

   ```
   char a = 65;
   char b = 'G';
   ```

   Operations can be performed on *char* variables, however, a typecast must be used in order for the result to also be a *char*:

   ```
   char a = 65;
   char b = 'G';
   char c = (char) b + c;
   ```

3. **A *boolean* value can have any value you like because any non-zero value is true. True or False?**

   In Java, a *boolean* value can only have two possible values: *true* or *false*.

4. **Given the following output, using a single string, show the *println()* statement that produced it.**

   ```
   One
   Two
   Three
   ```

   This output can be created using the following code:

   ```
   System.out.println("One\nTwo\nThree");
   ```

5. **What is wrong with this fragment?**

   ```
   for (i=0; i<10; i++){
     int sum;
     sum = sum + i;
   }

   System.out.println("Sum is: " + sum);
   ```

   This code will not compile because the *sum* variable is referred to outside its *scope*. Once the code block in which *sum* is declared has ended, *sum* will be designated for garbage collection and not available to the rest of the program. The corrected code would look like this:

   ```
   int sum;

   for (i=0; i<10; i++){
     sum = sum + i;
   }

   System.out.println("Sum is: " + sum);
   ```

6. **Explain the difference between the prefix and the postfix forms of the increment operator.**

   The prefix and postfix operators look like this:

```
int a = 10;
++a; // prefix form of increment operator
a++; // postfix form of increment operator
```

   There is a difference in how they work. The prefix operator will add "1" to the value of *a* **before** its value is assigned to another variable. The postfix version will increment the value of *a* **after** it has performed any other assignments needed first. As an example:

```
int a = 10;
int c;

c = ++a; // c will equal 11, and a will equal 11

a = 10; // reset a to 10
c = a++; // c will equal 10, and a will equal 11
```

7. **Show how a short-circuit AND can be used to prevent a divide-by-zero error.**

   A divide-by-zero error will cause any program to encounter an error. These (and other errors) can be avoided by using "short-circuit logical operators," which will evaluate the first operand of a statement and will only evaluate the second operand if it is deemed necessary. In the case of logical short circuit AND (&&) it will skip evaluating the second operand if the first is false, because if one operand is false then the whole statement must be false. In the case of a logical short circuit OR (||), it will skip evaluating the second operand if the first operand is true, since the statement is true if either operand is true. As an example (taken from text with slight modifications):

```
class SCops {
  public static void main(String[] args){

  int n,d,q;

  n = 10;
  d = 0;

    if (d != 0 && (n % d) == 0) { // second operand
                                  // never evaluated if d=0
      System.out.println(d + " is a factor of " + n);
    }
  } // end main
} // end SCops
```

8. **In an expression, what type are *byte* and *short* promoted to?**

   Java performs an *automatic type conversion* on variables if they are involved in an operation where the result is larger than the source type. If a *byte* or a *short* are involved in operations, they are automatically type converted to an *int*. For example:

```
byte b = 12;
short c = 193;

c = b + c; // c would be automatically cast as an int
```

9. **In general, when is a cast needed?**

   The programmer must perform a cast when he/she wants to intentionally restrict the result of an operation to a specific data type, even though this could result in a truncation of the results. Consider:

```
double d = 23.47;
double e = 34.58;
int f;

f = (int)d * e; // the result would be the integer,
                // stripped of decimal places.
```

10. **Write a program that finds all of the prime numbers between 2 and 100**

    The following program is from the supplied answers, with a few modifications. It was tested up to 1,000,000,000 and works perfectly:

```
class Prime {

  public static void main(String[] args){
    int i, j;
    int n = 100;  // max value
    boolean isPrime;

    for (i=2; i < n; i++){
      isPrime = true;

      // see if the number is evenly divisible
      for (j=2; j <= i/j; j++)
        // if it is, then it's not prime
        if ((i%j) == 0){
          isPrime = false;
        } // end if

        // if prime, print
        if (isPrime){
          System.out.print(i + " is prime.\t");
        }// end if

    }// end for
  } // end main
} // end Prime
```

11. **Does the use of redundant parentheses affect program performance?**

    Schildt states on Page 60 "Use of redundant or additional parentheses will not cause errors or slow down the execution of the expression." As always, it is a really good idea to make your code easily readable by humans (yourself and other programmers) because it is likely that someone will have to review the code if (when) there is a problem. Make your code easy to read - it is more important that the code be debuggable than the code being free of extra characters/spaces.

12. **Does a block define a scope?**

    Yes. A code block is the definition of *scope*. Each new code block defines a new scope. Nested code blocks are within the scope of the parent code block, but the parent is not within the scope of the nested child. Code blocks determine the scope and lifespan of objects.