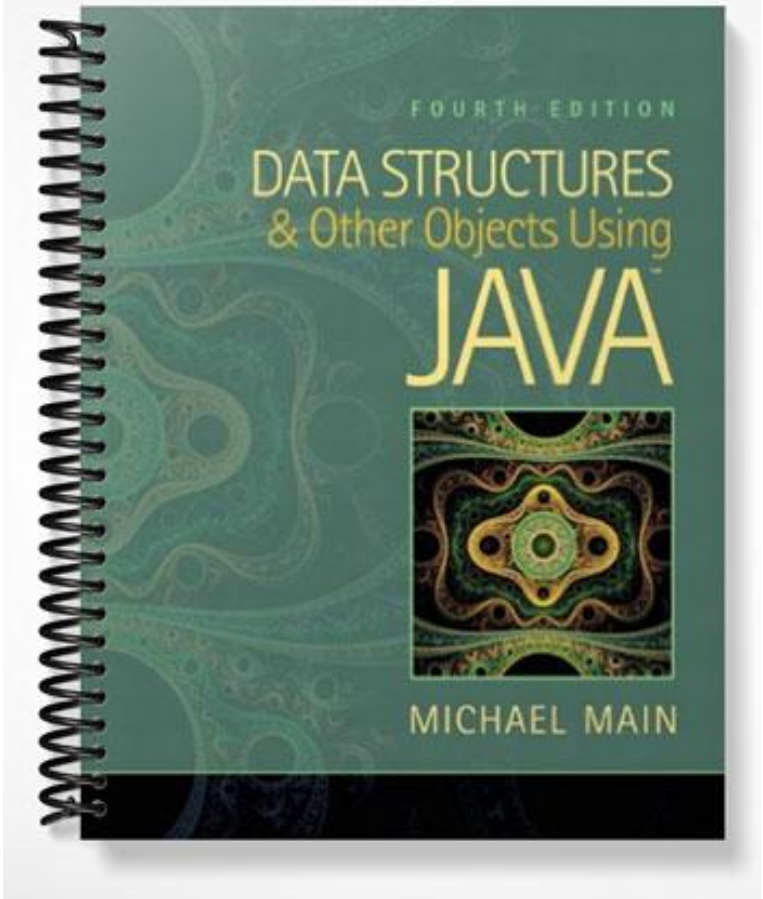
SOLUTIONS MANUAL



Weiss 4th Edition Solutions to Exercises (US Version)

Chapter 1 – Primitive Java

1.1 Key Concepts and How To Teach Them

This chapter introduces primitive features of Java found in all languages such as Pascal and C:

- basic lexical elements
- primitive types
- basic operators
- control flow
- functions (known as methods in Java)

Students who have had Java already can skip this chapter. I teach the material in the order presented. There is little tricky material here (this is part of the appeal of Java). Although the text does not mention C or C++, here are some differences:

1. Primitive types have precise ranges. There is no unsigned type. A char is 16 bits.
2. Order of evaluation is guaranteed (generally left to right). In particular, the sequence point rule from C++ is not needed. Thus nonsense such as x++ + x++ has a precise behavior in Java.
3. Only the C-style type conversion is allowed.
4. boolean is a primitive type, thus removing many of the common errors in C++, such as if(x=y) ... .
5. There is no comma operator, except in for loop expressions.
6. Java provides a labeled break statement.
7. 7. All functions must be class methods.

1.2 Solutions to Exercises

IN SHORT

1.1 Java source files end in `.java`. Compiled files (containing j-code or byte-codes) end in `.class`.

1.2 `//`, which extends to the end of the line and `/*` and `/**`, both of which extend to a `*/`. Comments do not nest.

1.3 `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, and `double`.

1.4 * multiplies two primitive values, returning the result and not changing its two arguments. *= changes the left-hand argument to the product of the left-hand argument and the right-hand argument. The right-hand argument is unchanged.

1.5 Both the prefix and postfix increment operators add one to the target variable. The prefix operator uses the new value of the variable in a larger expression; the postfix operator uses the prior value.

1.6 The `while` loop is the most general loop and performs a test at the top of the loop. The body is executed zero or more times. The `do` loop is similar, but the test is performed at the bottom of the loop; thus the body is executed at least once. The `for` loop is used primarily for counting-like iteration and consists of an initialization, test, and update along with the body.

1.7 `break` is used to exit a loop. A labeled `break` exits the loop that is marked with a label. `break` is also used to exit a `switch` statement, rather than stepping through to the next case.

1.8 The `continue` statement is used to advance to the next iteration of the loop.

1.9 Method overloading allows the reuse of a method name in the same scope as long as the signatures (parameter list types) of the methods differ.

1.10 In call-by-value, the actual arguments are copied into the method's formal parameters. Thus, changes to the values of the formal parameters do not affect the values of the actual arguments.

IN THEORY

1.11 After line 1, `b` is 6, `c` is 9, and `a` is 13. After line 2, `b` is 7, `c` is 10, and `a` is 16. After line 3, `b` is 8, `c` is 11, and `a` is 18. After line 4, `b` is 9, `c` is 12, and `a` is 21.

1.12 The result is `true`. Note that the precedence rules imply that the expression is evaluated as `(true && false) || true`.

1.13 The behavior is different if `statements` contains a `continue` statement.

1.14 Because of call-by-value, `x` must be 0 after the call to method `f`. Thus the only possible output is 0.

IN PRACTICE

1.15 An equivalent statement is:

```
while( true )
      statement
```

1.16 This question is harder than it looks because I/O facilities are limited, making it difficult to align columns.

```
public static void addition()
{
   for( int i = 0; i < 10; i++ )
   {
      for( int j = 0; j < 10; j++ )
      {
         if( i + j < 10 )
         System.out.print( " " );
         System.out.print( i + j + " " );
      }
   System.out.println( );
   }
}
```

```
public static void multiplication()
{
   for( int i = 1; i < 10; i++ )
   {
      for( int j = 1; j < 10; j++ )
      {
         if( i * j < 10 )
               System.out.print( " " );
         System.out.print( j * i + " " );
      }
   System.out.println( );
   }
}
```

1.17 The methods are shown below (without a supporting class); we assume that this is placed in a class that already provides the max method for two parameters.

```
public static int max( int a, int b, int c )
```

```
{
   return max( max( a, b ), c );
}
```

```
public static int max( int a, int b, int c, int d )
{
   return max( max( a, b ), max( c, d ) );
}
```

1.18 The method is below:

```
public static boolean isLeap( int year )
{
   boolean result = year % 4 == 0 &&
      ( year % 100 != 0 || year % 400 == 0 );

   return result;
}
```

1.3 Exam Questions

1.1 Consider the following statements:

```
int a = 4;
int b = 7;
b *= a;
```

What are the resulting values of `a` and `b`?

a. a is 4, b is 7
b. a is 28, b is 7
c. a is 4, b is 28
d. a is 28, b is 28
e. the statement is illegal

1.2 Consider the following statements:

```
int a = 4;
int b = 7
int c = ++a + b—;
```

What is the resulting value of `c`?

a. 10
b. 11
c. 12
d. 13
e. none of the above

1.3 Consider the following statements:

```
boolean a = false;
boolean b = true;
boolean c = false;
boolean d;
```

In which of the following is `d` evaluated?

a. `a && d`
b. `b && d`

c. `b || d`
d. `c && d`
e. All the operations evaluate `d`

1.4 Which of the following loop constructs guarantee that the body is executed at least once?
a. do loop
b. for loop
c. while loop
d. two of the constructs
e. all three of the constructs

1.5 In the method below, which statement about the possible outputs is most correct?

```
public static void what( )
{
   int x = 5;
   f( x );
   System.out.println( x );
}
```

a. 0 is a possible output
b. 5 is the only possible output
c. any positive integer can be output
d. any integer can be output
e. none of the above are true

1.6. If two methods in the same class have the same name, which is true:
a. They must have a different number of parameters
b. They must have different return types
c. They must have different parameter type lists
d. The compiler must generate an error message
e. none of the above

1.7 Consider the following statements:

```
int a = 5;
int b = 7;
int c, d;
c = (b - a)/2*a;
d = b + 7/a;
```

What are the resulting values of `c`
a. `c` is 0 and `d` is 2
b. `c` is 0 and `d` is 2.4
c. `c` is 5 and `d` is 8
d. `c` is 5 and `d` is 2
e. none of the above

1.8 Which of the following is true and `d`?
a. A variable must be declared immediately before its first use.
b. An identifier may start with any letter or any digit.
c. _33a is a valid identifier.
d. both (a) and (c) are true
e. all of the above are true

Answers to Exam Questions

1. C
2. C
3. B
4. A
5. B
6. C
7. C
8. C

Chapter 2 – Reference Types

2.1 Key Concepts and How To Teach Them

This chapter introduces several concepts:

- references
- strings
- arrays
- exceptions
- I/O

Depending on the students' background, some or even this entire chapter could be skipped, but I recommend at least a quick review of the chapter in all circumstances. Students who have not had Java should go through the entire chapter slowly because there are fundamental differences between Java objects and objects in other languages.

2.1.1 References

Explain the general idea of a reference in the context of a pointer. This is important to do because pointers are fundamental in most other languages. Then discuss the basic operations. Most important is to explain that objects must be created via new, what = and == means for references, and parameter passing for reference types. Students that have used other languages may be confused with the distinction between call-by-reference in a language such as Pascal, C++, or Ada and call-by-value applied to reference types in Java. Emphasize that the state of the referenced object can change, but the object being referenced by the actual argument prior to the method call will still be referenced after the method call.

2.1.2 Strings

Explain the basics of the `String`; especially that it is not an array of characters. The tricky parts are mixing up == and `equals`, and remembering that the second parameter to `subString` is the first non-included position.

2.1.3 Arrays

The tricky part about arrays are that typically the array must be created via `new` and if the array is an array of `Object`, then each `Object` in the array must also be created by `new`. Also, array copies are shallow.

2.1.4 Dynamic Expansion

Explain why we double instead of simply add an extra position. Discuss `ArrayList` as a type that maintains the size as well as capacity of an array and automatically increases capacity when needed.

2.1.5 Exceptions

Designing inheritance hierarchies is deferred to Chapter 4 when inheritance is discussed. This section simply discusses the `try`, `catch`, `finally` blocks, the `throw` clause and the `throws` list. Students do not seem to have difficulty with this material.

2.1.6 Input and Output

This section gives a brief description of I/O (mostly I), and the `StringTokenizer`. The I/O uses Java 1.1 constructs. This accounts for almost all of the updating from Java 1.0. The `StringTokenizer` is extremely important for simplifying the parsing of input lines. Without it, life will be difficult.

2.2 Solutions To Exercises

IN SHORT

2.1 Reference values (logically) store the address where an object resides; a primitive value stores the value of a primitive variable. As a result, operations such as == have seemingly different meanings for reference types and primitive types.

2.2 The basic operations that can be applied to a reference type are assignment via =, comparison via == and `!=`, the dot operator, type conversion, and `instanceof`.

2.3 An array has its size associated with it. An `ArrayList` has a capacity in addition to size. Adding an element to an `ArrayList` will automatically expand the capacity of the array if needed.

2.4 Exceptions are thrown by a method. The exception immediately propagates back through the calling sequence until it is handled by a matching `catch` clause, at which point the exception is considered handled.

2.5 Basic string operations include `equals` and `compareTo` to compare, = to copy, + and += to perform concatenation, `length`, `charAt`, and `substring`.

2.6 The `next` method returns the next token from the `Scanner` object, while the `hasNext` method returns a `true` value if there exists a token available to be read on the `Scanner`'s stream.

IN THEORY

2.7 The second statement outputs 5 7, as expected. The first outputs 44, because it used the ASCII value of ' ', which is 32.

2.8 The source code in Figure 2.21 fails to compile as shown. In the `foo` method, the compiler does not permit the multiple return statements (additionally the method is `void`). If you execute a call to the `bar` method, a `java.lang.ArithmeticException` is thrown.

IN PRACTICE

2.9 One possible solution to accomplish this is:

```
import java.io.*;
import java.util.*;

public class Checksum
{
   public static void main(String[] args)
   {
      int checksum = 0;
      System.out.print("Enter the name of the file: ");
      String filename = (new Scanner(System.in)).nextLine();
      Scanner fileScanner = null;
      try
      {
         fileScanner = new Scanner(new File(filename));
      } catch (IOException ioe)
      {
         System.err.println("IO Exception thrown.");
      }

      while (fileScanner.hasNextLine())
      {
         String line = fileScanner.nextLine();
         for (int i=0; i < line.length(); i++)
            checksum += line.charAt(i);
      }
      System.out.println("File checksum = " + checksum);
   }
}
```

2.10 One possible solution to do this is:

```
import java.util.Scanner;
```

```
import java.io.FileReader;
import java.io.IOException;

public class ListFiles
{
   public static void main( String [ ] args )
   {
      Scanner scanIn = null;
      if( args.length != 0 )
      {
         for( String fileName : args )
         {
            System.out.println( "FILE: " + fileName );
            try
            {
               scanIn = new Scanner (new FileReader ( fileName ));
               listFile( scanIn );
            }
            catch( IOException e )
            {
               System.out.println( e );
            }
            finally
            {
               // Close the stream
               if( scanIn != null )
                        scanIn.close( );
            }
         }
      } else
      {
         scanIn = new Scanner (System.in);
         listFile( scanIn );

         // Close the stream
         if( scanIn != null )
               scanIn.close( );
      }
   }

   public static void listFile( Scanner fileIn )
   {
      while( fileIn.hasNextLine( ) )
      {
            String oneLine = fileIn.nextLine( );
            System.out.println( oneLine );
      }
   }
}
```

2.11 A method to do this is below:

```
public static boolean isPrefix( String str1, String str2 )
{
   if( str1.length( ) > str2.length( ) )
      return false;
```

```
   for( int i = 0; i < str1.length( ) ; i++ )
      if( str1.charAt( str1.length() ) !=
         str2.charAt( str2.length() ) )
         return false;

   return true;
}
```

2.12 A method to do this is below:

```
public static int getTotalStrLength(String [] theStrings )
{
   int total = 0;
   for( String s : theStrings )
      total += s.length( );
   return total;
}
```

2.13 The elements in the original array are not copied before it is reinitialized.

2.14 Methods to accomplish this are below:

```
public static double sum( double [ ] arr )
{
   double sum = 0.0d;
   for (int i = 0; i < arr.length; i++ )
      sum += arr[i];
   return sum;
}
```

```
public static double average( double [ ] arr )
{
   double sum = 0.0d;
   for (int i = 0; i < arr.length; i++ )
      sum += arr[i];
   return sum / arr.length;
}
```

```
public static double mode( double [ ] arr )
{
   java.util.Arrays.sort(arr);

   int modeCounter = 1;
   int maxMode = 1;
   double modeValue = arr[0];

   for (int i = 0; i < arr.length-1; i++ )
   {
      if (arr[i] == arr[i+1])
         modeCounter++;

      if (modeCounter > maxMode) {
            maxMode = modeCounter;
            modeValue = arr[i+1];
```

```
            modeCounter = 1;
      }
   }
   return modeValue;
}
```

2.15 Methods to accomplish this are below:

```
public static double sum( double [ ][ ] arr )
{
   double sum = 0.0d;
   for (int row = 0; row < arr.length; row++ )
      for (int col = 0; col < arr[row].length; col++ )
         sum += arr[row][col];
   return sum;
}
```

```
public static double average( double [ ][ ] arr )
{
   double sum = 0.0d;
   int row = 0, col = 0;
   for (row = 0; row < arr.length; row++ )
      for (col = 0; col < arr[row].length; col++ )
         sum += arr[row][col];
   return sum / (row*col);
}
```

```
public static double mode( double [ ][] arr )
{
   double [] newArr = new double[arr.length * arr[0].length];
   int counter = 0;
   for (int row = 0; row < arr.length; row++ )
      for (int col = 0; col < arr[row].length; col++ )
         newArr[counter++] = arr[row][col];

   java.util.Arrays.sort(newArr);

   int modeCounter = 1;
   int maxMode = 1;
   double modeValue = newArr[0];

   for (int i = 0; i < newArr.length-1; i++ )
   {
      if (newArr[i] == newArr[i+1])
         modeCounter++;

      if (modeCounter > maxMode) {
         maxMode = modeCounter;
         modeValue = newArr[i+1];
         modeCounter = 1;
      }
   }
   return modeValue;
}
```