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;
}
```

2.16 Methods to accomplish this are below:

```
public static void reverse( String [ ] arr )
{
   for (int i=0; i < arr.length / 2; i++)
   {
      String temp = arr[arr.length-i-1];
      arr[arr.length-i-1] = arr[i];
      arr[i] = temp;
   }
}
```

```
public static void reverse( ArrayList<String> arr )
{
   for (int i=0; i < arr.size() / 2; i++)
   {
      String temp = arr.get(arr.size()-i-1);
      arr.set(arr.size()-i-1, arr.get(i));
      arr.set(i, temp);
   }
}
```

2.17 Methods to accomplish this are below:

```
public static int min( int [ ] arr )
{
   int currentMin = arr[0];
   for (int value : arr)
      if (value < currentMin)
         currentMin = value;
   return currentMin;
}
```

```
public static int min( int [ ][ ] arr )
{
   int currentMin = arr[0][0];
   for (int row = 0; row < arr.length; row++)
      for (int col = 0; col < arr[row].length; col++)
         if (arr[row][col] < currentMin)
            currentMin = arr[row][col];
   return currentMin;
}
```

```
public static String min( String [ ] arr )
{
   String currentMin = arr[0];
   for (String value : arr)
      if (value.compareTo(currentMin) < 0)
         currentMin = value;
   return currentMin;
}
```

```
public static String min( ArrayList<String> arr )
{
   String currentMin = arr.get(0);
   for (String value : arr)
      if (value.compareTo(currentMin) < 0)
         currentMin = value;
   return currentMin;
}
```

2.18 A method to do this is below:

```
public static int rowWithMostZeros( int [ ] [ ] arr )
{
   int result = 0, maxCount = 0;
   for (int row = 0; row < arr.length; row++)
   {
      int rowCounter = 0;
      for (int col = 0; col < arr[row].length; col++)
         if (arr[row][col] == 0)
            rowCounter++;
      if (rowCounter > maxCount)
      {
         maxCount = rowCounter;
         result = row;
      }
   }
   return result;
}
```

2.19 Methods to accomplish this are below:

```
public static boolean hasDuplicates( int [ ] arr )
{
   boolean result = false;
   for (int i=0; i < arr.length-1; i++)
      for (int j=i+1; j < arr.length; j++)
         if (arr[i] == arr[j])
            result = true;
   return result;
}
```

```
public static boolean hasDuplicates( int [ ][ ] 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);

   boolean result = false;
   for (int i=0; i < newArr.length-1; i++)
      for (int j=i+1; j < newArr.length; j++)
         if (newArr[i] == newArr[j])
```

```
            result = true;
   return result;
}
```

```
public static boolean hasDuplicates( String [ ] arr )
{
   boolean result = false;
   for (int i=0; i < arr.length-1; i++)
      for (int j=i+1; j < arr.length; j++)
         if (arr[i].compareTo(arr[j]) == 0)
            result = true;
   return result;
}
```

```
public static boolean hasDuplicates( ArrayList<String> arr )
{
   boolean result = false;
   for (int i=0; i < arr.size()-1; i++)
      for (int j=i+1; j < arr.size(); j++)
         if (arr.get(i).compareTo(arr.get(j)) == 0)
            result = true;
   return result;
}
```

2.20 Methods to accomplish this are below:

```
public static int howMany( int [ ] arr, int val )
{
   int result = 0;
   for (int i = 0; i < arr.length; i++)
      if (arr[i] == val)
         result++;
   return result;
}
```

```
public static int howMany( int [ ][ ] arr, int val )
{
   int result = 0;
   for (int i = 0; i < arr.length; i++)
      for (int j = 0; j < arr.length; j++)
         if (arr[i][j] == val)
            result++;
   return result;
}
```

2.21 Methods to accomplish this are below:

```
public static int countChars( String str, char ch )
{
   int counter = 0;
   for (int i = 0; i < str.length(); i++)
      if (str.charAt(i) == ch)
         counter++;
```

```
   return counter;
}
```

```
public static int countChars( String [ ] str, char ch )
{
   int counter = 0;
   for (int stringNum = 0; stringNum < str.length; stringNum++)
      for (int i = 0; i < str[stringNum].length(); i++)
         if (str[stringNum].charAt(i) == ch)
            counter++;
   return counter;
}
```

2.22 Methods to accomplish this include:

```
public static String [ ] getLowerCase( String [ ] arr )
{
   String [] results = new String[arr.length];
   int index = 0;
   for (String value : arr)
      results[index++] = value.toLowerCase();
   return results;
}
```

```
public static void makeLowerCase( String [ ] arr )
{
   int index = 0;
   for (String value : arr)
      arr[index++] = value.toLowerCase();
}
```

```
public static ArrayList<String> getLowerCase( ArrayList<String> arr )
{
   ArrayList<String> results = new ArrayList<String>();
   for (String value : arr)
      results.add(value.toLowerCase());
   return results;
}
```

```
public static void makeLowerCase( ArrayList<String> arr )
{
   int index = 0;
   for (String value : arr)
      arr.set(index++, value.toLowerCase());
}
```

2.23 One solution is:

```
public static boolean isIncreasing( int [ ] [ ] arr )
{
   boolean increasing = true;
   for (int row = 0; row < arr.length; row++)
```

```
      for (int col = 0; col < arr[row].length-1; col++)
         if (arr[row][col] > arr[row][col+1])
            increasing = false;
   return increasing;
}
```

2.24 One possible solution is:

```
public ArrayList<String> startsWith( String [ ] arr, char ch )
{
   ArrayList<String> results = new ArrayList<String>();
   for (String value : arr)
      if (value.charAt(0) == ch)
         results.add(value);
   return results;
}
```

2.25 One possible solution is:

```
public String [ ] split( String str )
{
   ArrayList<String> list = new ArrayList<String>();
   Scanner scan = new Scanner(str);
   while (scan.hasNext())
      list.add(scan.next());
   return ((String []) list.toArray());
}
```

2.26 One possible solution includes:

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

class MaxTestA
{
   public static void main( String [ ] args )
   {
      Scanner in = new Scanner( System.in );
      int x, y;

      System.out.println( "Enter 2 ints: " );
      String inputLine = in.nextLine();
      String [] inputs = inputLine.split("\\s");

      x = new Integer(inputs[0]).intValue();
      y = new Integer(inputs[1]).intValue();
      System.out.println( "Max: " + Math.max( x, y ) );

      return;
   }
}
```

2.27 Here is one possible solution. For whatever reason, reading directly from the input stream (`System.in`) and using the specified delimiter would not work (tried on multiple platforms). Was forced to read the input line as a `String` and then use a second scanner on the `String` to break it into tokens using the delimiter.

```
public static void MaxTestA( )
{
   Scanner in = new Scanner( System.in );
   int x, y;

   System.out.println( "Enter 2 ints separated by a comma" );
   String inputLine = in.nextLine();
   Scanner scan = new Scanner(inputLine);
   scan.useDelimiter("[,]");
   if (scan.hasNextInt())
   {
      x = scan.nextInt();
      if (scan.hasNextInt())
      {
         y = scan.nextInt();
         System.out.println( "Max: " + Math.max( x, y ) );
         return;
      }
   }
   System.err.println("Error: need two ints separated by a comma" );
   return;
}
```

2.3 Exam Questions

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

```
public static void what( )
{
   Integer x = new Integer( 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

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

```
public static void what( )
{
   Integer x;
   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

2.3 Which class is used to parse a line of input?

a. `BufferedReader`
b. `FileReader`

c. `InputStreamReader`
d. `StringTokenizer`
e. none of the above

2.4 Which of the following is not true about a `String`?
a. `String`s are reference types.
b. Individual characters can be accessed.
c. `String`s must be created without using `new`.
d. The contents of a `String` can be safely copied via =.
e. The length of a `String` can always be determined.

2.5 Which of the following is not true about arrays?
a. Arrays are reference types.
b. Array indexing is bounds-checked.
c. Arrays sometimes can be created without using `new`.
d. The entire contents of an array can be copied via =.
e. The capacity of an array can always be determined.

2.6 Which of the following is true about reference types?
a. They are initialized to 0 by default.
b. = can be used to copy the states of two objects.
c. == can be used to test if the referenced objects have identical states.
d. They are passed using call-by-reference.
e. all of the above are false

2.7 Which of (a) - (d) is false:
a. A `catch` block for a standard run-time exception is optional.
b. An exception is an object
c. A `throws` clause is used to throw an exception.
d. An exception handler is inside a `catch` block.
e. all of the above are true

2.8 Which of the following is false about type `ArrayList`:
a. The `add` operation always increase size by 1.
b. Only objects can be stored in an `ArrayList`
c. An `add` operation may increase the capacity by more than one.
d. The capacity of an `ArrayList` can be changed using the `set` method.
e. none of the above is false

Answers to Exam Questions

1. B
2. E
3. D
4. C
5. D
6. E
7. C
8. D

Chapter 3 – Objects and Classes

3.1 Key Concepts and How To Teach Them

Students who have not had Java, with a description of class design, will need to go through this chapter in its entirety. Students who have had Java with class design may want to quickly review the chapter. This chapter introduces the general concept of encapsulation and information hiding, but is geared towards practical use of Java with emphasis on designing classes and syntax. You may want to have the students bring a copy of the code to class so you can avoid rewriting the classes. This chapter is by far much simpler than its C++ counterpart.

Topics include:

- the class construct
- `public` and `private` sections
- specification (javadoc) vs. implementation
- constructors
- accessors and mutators
- `toString`
- `equals`
- packages
- `this`
- `instanceof` operator
- `static` class members

3.1.1 The class Construct

Describe how the class achieves the grouping of data members and the information hiding and encapsulation of functionality. The basic mechanism for the latter is to allow methods to be class members. You can illustrate this with MemoryCell. Add the `private` and `public` visibility modifiers after discussing it.

3.1.2 Public and Private Sections

This seems to be a relatively easy topic. Explain that everything in the private section is inaccessible to non-class routines. Continue with the MemoryCell example.

3.1.3 Specification vs. Implementation

Explain the importance of the specification, and how some of it can be generated automatically via javadoc.

3.1.4 Constructors

Explain that in addition to particular member functions, every class has constructors. Show how a constructor matches a declaration. Remark about the default behavior: If no constructor is provided, a default zero-parameter constructor is generated.

3.1.5 Accessors and Mutators

Java has no formal way to specify an accessor or a mutator. However the concept is important to teach.

3.1.6 toString

This is a simple topic to cover.

3.1.7 equals

The tricky part is that the parameter is always of type `Object` and must be converted to the class type. Mention that if `equals` is not implemented, then it typically defaults to returning false (actually it is inherited from its superclass).

3.1.8 Packages

Mention why we use packages and the special visibility rules. Also discuss the `import` directive briefly. This is a reasonable time to discuss compilation issues and the CLASSPATH variable.

3.1.9 this

`this` is used in two main places: aliasing tests (in which it is important to have a reference to the current object), and as a shorthand for calling other constructors. Both uses are easy to discuss.

3.1.10 instanceof Operator

This is used mostly in the `equals` method. Note that the `null` reference is not an instance of any class.

3.1.11 static Class Members

Explain that a static field is unique to a class and does not belong to any instance of that class. They can be referenced via class names or object references.

3.1.12 Design patterns

Just mention that design patterns are used to encode commonly occurring problems and their solutions, and that we will be identifying design patterns in the course.

3.2 Solutions To Exercises

IN SHORT

3.1 Information hiding makes implementation details, including components of an object, inaccessible. Encapsulation is the grouping of data and the operations that apply to them to form an aggregate while hiding the implementation of the aggregate. Encapsulation and information hiding are achieved in Java through the use of the class.

3.2 Members in the public section of a class are visible to non-class routines and can be accessed via the dot member operator. Private members are not visible outside of the class.

3.3 The constructor is called when an object is created by a call to `new`.

3.4 The default constructor is a member-by-member application of a default constructor, meaning that primitive members are initialized to zero and reference members are initialized to null.

3.5 `this` is a reference to the current object. This reference to the current object can be used to compare it with other objects or to pass the current object to some other unit.

3.6 The constructor would be viewed as a method, and would not be treated as a constructor.

3.7 Packages are used to organize similar classes. Members with no visibility modifier are package visible: that is, they are visible to other classes within the same package (it is not visible outside the package).

3.8 Output is typically performed by providing a `toString` method that generates a `String`. This `String` can be passed to `println`. However, a nonstatic field, which is part of each instance of the class, can be accessed by a static class method only if a controlling object is provided.

3.9 The two import directives are below:

```
import weiss.nonstandard.*;
import weiss.nonstandard.Exiting;
```

3.10 An instance field is associated with a particular instance (or object) of a class. A static field is associated with the class itself and can be accessed from any instance of the class.

3.11 A `static` method can be invoked without any instance (and therefore any instance data) existing. Even if several instances did exist, there'd be no way of knowing which object's instance data was being referenced.

3.12 A design pattern describes a commonly occurring problem in many contexts and a generic solution that can be applied in a wide variety of these contexts.

3.13 (a) Line 17 is illegal because `p` is a non-static field and therefore needs an object reference in order to be reference in the `main` method (which is a static method). Line 18 is legal as `q` is created within the `main` method. (b) Line 20 and 23 are legal as `NO_ SSN` is a public static field that can be accessed via an object reference or a class name. Line 22 is legal because by default, `name` is visible. Line 21 and 24 are illegal because `SSN` is private. Also, `Person.SSN` is illegal as `SSN` is nonstatic.

IN THEORY

3.14 By defining a single private constructor for a class A, we can disallow any other object to create an instance of A. For example, we may use A to hold only static data.

3.15 (a) Yes; `main` works anywhere. (b) Yes; if `main` was part of class `IntCell`, then `storedValue` would no longer be considered private to `main`.

3.16 No, only one wildcard symbol (`*`) can be used in an import statement and only to specify all classes within a specified package (i.e. `java.util.*`).

3.17 A run time error will occur stating that the zero-parameter constructor could not be found.

```
Exception in thread "main" java.lang.NoSuchMethodError: IntCell: method
<init>()V not found
   at TestIntCell.main(TestIntCell.java:7)
```

IN PRACTICE

3.18 One possible solution is:

```
public class Combolock
{
   private int num1;
   private int num2;
   private int num3;

   public Combolock ( int a, int b, int c)
   {
      num1 = a;
      num2 = b;
      num3 = c;
   }

   //returns true if the proper combination is given
   public boolean open ( int a, int b, int c)
   {
      return ( ( a == num1 ) && ( b == num2 ) && (c == num3 ) );
   }

   public boolean changeCombo( int a, int b, int c, int newA,
         int newB, int newC )
   {
      if ( open( a, b, c) )
      {
         num1 = newA;
         num2 = newB;
         num3 = newC;
         return true;
      }
   return false;
}
```