

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. 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. `Strings` are reference types.
 - b. Individual characters can be accessed.
 - c. `Strings` 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;
    }
}
```

- 3.19 The surprising result in part (d) is that the compiler will find the bytecodes for the local `List` class, even though you have recommended it. And thus, no ambiguity will be reported. Even with the `import` directive, the local `List` class will win out over `java.awt.List`.
- 3.20 The modification to `TestIntCell` results in the addition of the `import` statement:

```
import weiss.nonstandard.IntCell;

// Exercise the IntCell class

public class TestIntCell
{
    public static void main( String [ ] args )
    {
        IntCell m = new IntCell( );

        m.write( 5 );
        System.out.println( "Cell contents: " + m.read( ) );
        // The next line would be illegal if uncommented
        // m.storedValue = 0;
    }
}
```

- 3.21 Implementations are as follows:

```
public BigRational pow( int exp ) // exception if exp<0
{
    BigRational result = this;
    if (exp < 0)
        throw new ArithmeticException( "EXPONENT < 0" );
    else
    {
        if (exp == 0)
            return BigRational.ONE;
        else if (exp == 1)
            return this;
        else for (int i = 2; i <= exp; i++)
        {
            result = result.multiply(this);
        }
    }
    return result;
}

public BigRational reciprocal( )
{
    BigRational newRational = new BigRational(den, num);
    newRational.fixSigns();
    return newRational;
}

public BigInteger toBigInteger( ) // exception if denominator is not 1
{
    if (!den.equals( BigInteger.ONE ))
        throw new ArithmeticException( "DENOMINATOR IS NOT ONE" );
    else
        return num;
}
```

```

}

public int toInteger( ) // exception if denominator is not 1
{
    if (!den.equals( BigInteger.ONE ))
        throw new ArithmeticException( "DENOMINATOR IS NOT ONE" );
    else
        return num.intValue();
}

```

3.22 The implementation is:

```

public BigRational( BigInteger n, BigInteger d )
{
    if( d.compareTo( BigInteger.ZERO ) < 0 )
        throw new ArithmeticException( " DENOMINATOR < 0" );
    if( num.equals( BigInteger.ZERO ) &&
        den.equals( BigInteger.ZERO ) )
        throw new IllegalArgumentException( "0/0" );
    num = n;
    den = d;
    reduce( );
}

```

3.23 A possible solution is:

```

package weiss.math;

import java.math.BigInteger;

public class BigRational2
{
    public static final BigRational2 ZERO = new BigRational2( );
    public static final BigRational2 ONE = new BigRational2( "1" );

    public BigRational2( String str )
    {
        if( str.length( ) == 0 )
            throw new IllegalArgumentException( "Zero-length string" );

        // Check for "/"
        int slashIndex = str.indexOf( '/' );
        if( slashIndex == -1 )
        {
            // no denominator... use 1
            den = BigInteger.ONE;
            num = new BigInteger( str.trim( ) );
        }
        else
        {
            den = new BigInteger( str.substring( slashIndex + 1 ).trim(
) );
            num = new BigInteger( str.substring( 0, slashIndex ).trim(
) );
            fixSigns( ); reduce( );
        }
    }
}

```

```

    }
}

// Ensure that the denominator is positive
private void fixSigns( )
{
    if( den.compareTo( BigInteger.ZERO ) < 0 )
    {
        num = num.negate( );
        den = den.negate( );
    }
}

// Divide num and den by their gcd
private void reduce( )
{
    if ( !den.equals(BigInteger.ZERO))
    {
        BigInteger gcd = num.gcd( den );
        num = num.divide( gcd );
        den = den.divide( gcd );
    }
}

public BigRational2( BigInteger n, BigInteger d )
{
    num = n;
    den = d;
    fixSigns( ); reduce( );
}

public BigRational2( BigInteger n )
{
    this( n, BigInteger.ONE );
}

public BigRational2( )
{
    this( BigInteger.ZERO );
}

public BigRational2 abs( )
{
    return new BigRational2( num.abs( ), den );
}

public BigRational2 negate( )
{
    return new BigRational2( num.negate( ), den );
}

public BigRational2 add( BigRational2 other )
{
    BigInteger newNumerator =
        num.multiply( other.den ).add(
            other.num.multiply( den ) );
    BigInteger newDenominator =

```

```

        den.multiply( other.den );

        return new BigRational2( newNumerator, newDenominator );
    }

    public BigRational2 subtract( BigRational2 other )
    {
        return add( other.negate( ) );
    }

    public BigRational2 multiply( BigRational2 other )
    {
        BigInteger newNumer = num.multiply( other.num );
        BigInteger newDenom = den.multiply( other.den );

        return new BigRational2( newNumer, newDenom );
    }

    public BigRational2 divide( BigRational2 other )
    {
        if( other.num.equals( BigInteger.ZERO ) && num.equals(
BigInteger.ZERO ) )
            throw new ArithmeticException( "ZERO DIVIDE BY ZERO" );

        BigInteger newNumer = num.multiply( other.den );
        BigInteger newDenom = den.multiply( other.num );

        return new BigRational2( newNumer, newDenom );
    }

    public boolean equals( Object other )
    {
        if( ! ( other instanceof BigRational2 ) )
            return false;

        BigRational2 rhs = (BigRational2) other;

        return num.equals( rhs.num ) && den.equals( rhs.den );
    }

    public String toString( )
    {
        if( num.equals( BigInteger.ZERO ) &&
            den.equals( BigInteger.ZERO ) )
            return "Indeterminate";

        if( den.equals( BigInteger.ZERO ) )
            if( num.compareTo( BigInteger.ZERO ) < 0 )
                return "-infinity";
            else
                return "infinity";

        if( den.equals( BigInteger.ONE ) )
            return num.toString( );
        else
            return num + "/" + den;
    }

```



```

        private BigInteger num; // only this can be neg
        private BigInteger den;
    }

```

3.24 One solution is:

```

import java.math.*;
import weiss.math.*;
import java.io.*;
import java.util.*;

public class RationalNumberTester
{
    public static void main (String [] args) throws
FileNotFoundException
    {
        ArrayList<BigRational> list = new ArrayList<BigRational>();
        Scanner scan = new Scanner(new File ("rationalNumbers.txt"));
        while (scan.hasNextLine())
        {
            String line = scan.nextLine();
            BigRational br = new BigRational(line);
            if (!listContains(list, br))
                list.add(br);
        }

        System.out.println("Sum: " + sum(list));
        System.out.println("Mean: " + (sum(list).divide(
            new BigRational(list.size()+"1"))));
        System.out.println("Recip. sum: " + reciprocalSum(list));
        System.out.println("Harmonic Mean: " +
            (new BigRational(list.size() +
                "/1")).divide(reciprocalSum(list)));
    }

    private static BigRational sum(ArrayList<BigRational> list)
    {
        BigRational sum = list.get(0);
        for (int i = 1; i < list.size(); i++)
            sum = sum.add(list.get(i));
        return sum;
    }

    private static BigRational reciprocalSum(ArrayList<BigRational>
list)
    {
        BigRational sum = list.get(0).reciprocal();
        for (int i = 1; i < list.size(); i++)
            sum = sum.add(list.get(i).reciprocal());
        return sum;
    }

    private static void printList(ArrayList<BigRational> list)
    {
        for (BigRational value : list)
            System.out.print(value + " ");
    }
}

```

```

        System.out.println();
    }

    private static boolean listContains(ArrayList<BigRational> list,
        BigRational item)
    {
        boolean result = false;

        for (int i = 0; !result && i < list.size(); i++)
            if (list.get(i).equals(item))
                result = true;

        return result;
    }
}

```

3.25 One possible solution is:

```

public static void printArray(int [][] arr)
{
    for (int row = 0; row < arr.length; row++)
    {
        for (int col = 0; col < arr[row].length; col++)
            System.out.print(String.format("%5d", arr[row][col]));
        System.out.println();
    }
}

```

3.26 (a) Yes, objects of the `BigDecimal` class are immutable. (b) With `compareTo`, two `BigDecimal` objects that are equal in value but have a different scale (like 2.0 and 2.00) are considered equal. When using the `equals` method, two `BigDecimal` objects are considered equal only if they are equal in value and scale. (c) `bd1.equals(bd2)` would be false when the value of `bd2` is also 0. (d) By default, 0.2 would be the result. (e) Since the result is a non-terminating decimal expansion, an `ArithmeticException` is thrown when executing the `divide` method. (f) An object with a precision setting matching the IEEE 754R Decimal128 format, 34 digits, and a rounding mode of `HALF_EVEN`, the IEEE 754R default. (g) One possible solution is:

```

package weiss.math;

import java.math.*;

public class BigRational3
{
    public static final BigRational3 ZERO = new BigRational3( );
    public static final BigRational3 ONE = new BigRational3( "1" );
    private MathContext mc = MathContext.UNLIMITED;

    public BigRational3( String str )
    {
        if( str.length( ) == 0 )
            throw new IllegalArgumentException( "Zero-length string" );

        // Check for "/"
        int slashIndex = str.indexOf( '/' );
        if( slashIndex == -1 )
        {

```

```

        // no denominator... use 1
        den = BigInteger.ONE;
        num = new BigInteger( str.trim( ) );
    }
    else
    {
        den = new BigInteger(
            str.substring( slashIndex + 1 ).trim( ) );
        num = new BigInteger(
            str.substring( 0, slashIndex ).trim( ) );
        check00( ); fixSigns( ); reduce( );
    }
}

private void check00( )
{
    if( num.equals( BigInteger.ZERO ) &&
        den.equals( BigInteger.ZERO ) )
        throw new IllegalArgumentException( "0/0" );
}

// Ensure that the denominator is positive
private void fixSigns( )
{
    if( den.compareTo( BigInteger.ZERO ) < 0 )
    {
        num = num.negate( );
        den = den.negate( );
    }
}

// Divide num and den by their gcd
private void reduce( )
{
    BigInteger gcd = num.gcd( den );
    num = num.divide( gcd );
    den = den.divide( gcd );
}

public BigRational3( BigInteger n, BigInteger d )
{
    num = n;
    den = d;
    check00( ); fixSigns( ); reduce( );
}

public BigRational3( BigInteger n )
{
    this( n, BigInteger.ONE );
}

public BigRational3( )
{
    this( BigInteger.ZERO );
}

public BigRational3 abs( )

```

```

    {
        return new BigRational3( num.abs( ), den );
    }

    public BigRational3 negate( )
    {
        return new BigRational3( num.negate( ), den );
    }

    public BigRational3 add( BigRational3 other )
    {
        BigInteger newNumerator =
            num.multiply( other.den ).add(
                other.num.multiply( den ) );
        BigInteger newDenominator =
            den.multiply( other.den );

        return new BigRational3( newNumerator, newDenominator );
    }

    public BigRational3 subtract( BigRational3 other )
    {
        return add( other.negate( ) );
    }

    public BigRational3 multiply( BigRational3 other )
    {
        BigInteger newNumer = num.multiply( other.num );
        BigInteger newDenom = den.multiply( other.den );

        return new BigRational3( newNumer, newDenom );
    }

    public BigRational3 divide( BigRational3 other )
    {
        if( other.num.equals( BigInteger.ZERO ) &&
            num.equals( BigInteger.ZERO ) )
            throw new ArithmeticException( "ZERO DIVIDE BY ZERO" );

        BigInteger newNumer = num.multiply( other.den );
        BigInteger newDenom = den.multiply( other.num );

        return new BigRational3( newNumer, newDenom );
    }

    public boolean equals( Object other )
    {
        if( ! ( other instanceof BigRational3 ) )
            return false;

        BigRational3 rhs = (BigRational3) other;

        return num.equals( rhs.num ) && den.equals( rhs.den );
    }

    public String toString( )
    {

```

```

        if( den.equals( BigInteger.ZERO ) )
            if( num.compareTo( BigInteger.ZERO ) < 0 )
                return "-infinity";
            else
                return "infinity";

        if( den.equals( BigInteger.ONE ) )
            return num.toString( );
        else
            return num + "/" + den;
    }

    public BigDecimal toBigDecimal()
    {
        BigDecimal decnum = new BigDecimal(num, mc);
        BigDecimal result = null;

        try
        {
            result = decnum.divide(new BigDecimal(den, mc), mc);
        }
        catch (ArithmeticException ae)
        {
            if (ae.getMessage().equals(
                "Non-terminating decimal expansion; " +
                "no exact representable decimal result."))
                result = decnum.divide(
                    new BigDecimal(den, mc), MathContext.DECIMAL128);
        }
        return result;
    }

    private BigInteger num;    // only this can be neg
    private BigInteger den;
}

```

3.27 One possible solution is below:

```

public class Account
{
    private float balance = 0.0f;

    public Account (float initAmt) throws Exception
    {
        if (initAmt < 0.0f)
            throw new Exception("Negative amounts are not possible.");
        balance = initAmt;
    }

    public float getBalance()
    {
        return balance;
    }

    public void deposit(float amt)
    {
        balance += amt;
    }
}

```

```

    }

    public void withdraw(float amt) throws Exception
    {
        if (amt < 0.0)
            throw new Exception("Negative amounts are not possible.");
        if (amt > balance)
            throw new Exception("Your balance can not be < 0.");
        balance -= amt;
    }

    public String toString()
    {
        return balance + "";
    }
}

```

3.28 One possible solution is as follows:

```

package weiss.misc;

public class BinaryArray
{
    private boolean [] arr = null;

    public BinaryArray(String init)
    {
        arr = new boolean[init.length()];
        for (boolean value : arr)
            value = false;

        for (int ch = 0; ch < init.length(); ch++)
        {
            boolean val = false;
            if (init.charAt(ch) == 'T')
                val = true;
            arr[ch] = val;
        }
    }

    public int size()
    {
        return arr.length;
    }

    public String toString()
    {
        String result = "";
        for (int i = 0; i < arr.length; i++)
            result += arr[i] + " ";
        return result;
    }

    public boolean get(int location)
    {
        return arr[location];
    }
}

```

```
public void set(int location, boolean value)
{
    arr[location] = value;
}
}
```

3.3 Exam Questions

- 3.1 Which of the following is the most direct example of how encapsulation is supported in Java?
- constructors
 - inheritance
 - methods
 - `public` and `private` specifiers
 - the class declaration
- 3.2 Which of the following is the most direct example of how information hiding is supported in Java?
- constructors
 - inheritance
 - member functions
 - `public` and `private` specifiers
 - the class declaration
- 3.3 What happens if a non-class method attempts to access a private member?
- compile time error
 - compile time warning, but program compiles
 - the program compiles but the results are undefined
 - the program is certain to crash
 - some of the above, but the result varies from system to system
- 3.4 Which condition occurs when the same object appears as both an input and output parameter?
- aliasing
 - copy construction
 - operator overloading
 - type conversion
 - none of the above
- 3.5 Which of (a) to (c) is false about a static class member?
- it must be a method
 - one member is allocated for each declared class object
 - the `static` class member is guaranteed to be `private` to the class
 - two of the above are false
 - all of (a), (b), and (c) are false
- 3.6 In which of the following cases is a class member *M* invisible to a method *F*?
- F* is a method in the same class and *M* is `private`
 - F* is a package friendly function and *M* is not `private`
 - F* is a method in another class and *M* is `public`
 - F* is a method in another class and *M* is `private`
 - none of the above
- 3.7 Which of the following statements is true?
- Any documentation produced by *javadoc* is guaranteed to be implemented by the class.

- b. Java provides a mechanism to distinguish accessors and mutators.
 - c. Each class can provide a `main` to perform testing.
 - d. `this` is available in all methods, including `static` methods.
 - e. Every class must implement `toString` and `equals`.
- 3.8 For class C, which are the parameter types of `equals` ?
- a. no parameters.
 - b. one parameter, type C.
 - c. one parameter, type `Object`.
 - d. two parameters, both of type C.
 - e. two parameters, both of type `Object`.
- 3.9 Which of (a) - (d) is false for the `main` method?
- a. Each class can have its own `main` method.
 - b. The arguments passed to a `main` method from a command line are strings.
 - c. The `main` method must be written as the last method in the class.
 - d. The `main` method typically does not return a value
 - e. all of the above are true
- 3.10 Which of the following is false for constructors?
- a. There may not be any constructor defined in a class.
 - b. A constructor must be declared `public`.
 - c. There may be multiple constructors in a class.
 - d. A constructor typically has no `return` value.
 - e. A constructor must have the same name as the class name.

Answers to Exam Questions

- 1. E
- 2. D
- 3. A
- 4. A
- 5. E
- 6. D
- 7. C
- 8. C
- 9. C
- 10. B

Chapter 4 - Inheritance

4.1 Key Concepts and How To Teach Them

Inheritance in Java is a fundamental concept. The text does not directly define large hierarchies, but repeatedly uses inheritance in two places: the interface and in designing generic algorithms. My recommendation is that other uses of inheritance should be occasional. The basic concepts are:

- The concept of inheritance and polymorphism
- Extending classes
- Designing hierarchies
- Abstract and final methods and classes; Static vs. dynamic binding
- Interfaces
- Generic components
- Adapters and wrappers

4.1.1 The Concept of Inheritance and Polymorphism

Inheritance is used when new classes are needed that have basic similarity to existing classes. Several examples are provided in the text. Use the `Person` example to explain the IS-A relation. Explain advantages of inheritance such as code reuse.

4.1.2 Extending Classes

I like to explain that the derived class is equal to the base class with possible additions (new data members, new methods) and modifications (redefinition of base class members). Explain that the derived class has all of the data members of the base class, plus its own. The inherited members are not directly accessible if they were `private` in the base class. Also explain that for the purposes of construction, the data members that form the base class are considered as an atomic object in the call to `super` (and if they are `private`, that is the only way to initialize the inherited portion of the class). Use very short examples, such as `Underflow`, initially. `Public`, `private`, and `protected` must be explained. This tends to not be a difficult concept.

4.1.3 Dynamic binding and polymorphism

A reference variable that is polymorphic can reference objects of several different types. When operations are applied to the reference, the operation that is appropriate to the actual referenced object is automatically selected (also referred to as dynamic binding). This concept is useful when derived classes may override base class methods. Explain the concept using the `Shape` example.

4.1.4 Designing hierarchies

Using the `Shape` example, explain the need for `abstract` methods, `final` methods and `static` methods. Similarly, explain the use of `abstract` classes and `final` classes. `Abstract` classes are placeholders and cannot be constructed. The derived classes must provide implementations for the `abstract` class methods. `Final` methods are those that cannot be overridden.

4.1.5 Interfaces

Interfaces are simply `abstract` classes with no implementation. In other words, they specify a protocol. They are syntactically simple and students do not seem to have trouble with the concept.

4.1.6 Inheritance in Java

Briefly explain the `Object` class and that all other class is a direct or indirect subclass of `Object`. The `Exception` and I/O hierarchies in the Java libraries are good examples to illustrate inheritance. Without going into too much detail (as some of these concepts are covered in later chapters), explain the exception hierarchy and I/O decorator pattern example for wrapping streams and readers to illustrate the I/O hierarchy.

4.1.7 Generic Components

A generic implementation gives the basic functionality in a type independent manner. The main idea is to use inheritance and write components in terms of the most general type of object that will be used (this component can then be reused in all derived classes). This can be accomplished by using an appropriate

superclass such as `java.lang.Object`. Use the `MemoryCell` example to illustrate a generic cell class. Interface types may sometimes be needed to define more specific operations than those available in the `Object` class. The `Comparable` interface can be used to write a generic `findMax` method. Java 1.5 has simplified the task of writing generic components with the introduction of language support for generic classes and methods. The generic mechanism in Java 1.5 is driven entirely by the compiler – essentially converting 1.5 generic code into a pre-1.5 inheritance-based generic implementation. For this reason, it is essential to understand inheritance based generic coding, even when working with Java 1.5.

4.1.8 Wrappers and Adapters

A Wrapper stores an object and adds operations, where as an adapter simply changes the interface without changing the functionality. Java 1.5 introduced new autoboxing and unboxing features that perform behind the scenes wrapping/unwrapping of primitive types with their corresponding class-based implementations.

4.1.9 Functor

A functor is typically a class with a single method that is used in a generic algorithm. Use the `findMax` example to illustrate the concept. Note that the algorithm typically specifies an interface class and the actual function object class must implement this interface.

4.1.10 Nested, anonymous and local classes

A nested class declaration is placed inside another class declaration. A local class is declared inside a method. Finally, an anonymous class is also declared inside a method but is declared immediately where it is used. Give a brief explanation using the `findMax` example. Students will get comfortable with them when they actually use them later in the course

4.2 Solutions To Exercises

IN SHORT

- 4.1 Assuming `public` inheritance, only `public` and `protected` members of the base class are visible in the derived class unless the derived class is in the same package of the base class, in which case package friendly members are also visible. Among base class members, only `public` members of the base class are visible to users of the derived class.
- 4.2 `Private` inheritance is used to indicate a HAS-A relationship while avoiding the overhead of a layer of function calls. Composition is a better alternative for this. In composition, a class is composed of objects of other classes.
- 4.3 Polymorphism is the ability of a reference type to reference objects of several different types. When operations are applied to the polymorphic type, the operation that is appropriate to the actual referenced object is automatically selected.
- 4.4 Autoboxing and unboxing are new features introduced in Java 1.5. They perform automatic conversion between primitive types and their object counterparts (e.g. `int` and `java.lang.Integer`).
- 4.5 A `final` method is a method that cannot be redefined in a derive class.
- 4.6 (a) Only `b.bPublic` and `d.dPublic` are legal. (b) if `main` is in `Base` , then only `d.dPrivate` is illegal. (c) If `main` is in `Derived`, then `b.bPrivate` is illegal. (d) Assuming `Tester` is in the same package, then `b.bProtect` is visible in all cases. (e) place the statements below in the `public` section of their respective classes; (f) and (g) `bPrivate` is the only member not accessible in `Derived`.

```
Base( int pub, int pri, int pro )
{
    bPublic = pub; bPrivate = pri; bProtect = pro;
}
```

```

Derived( int bpub, int bpri, int bpro, int dpub, int dpri )
{
    super( bpub, bpri, bpro );
    dPublic = dpub; dPrivate = dpri;
}

```

- 4.7 Final classes may not be subclassed (by the use of inheritance). Non-final classes can be subclassed. Final classes are generally used to provide an implementation of a class that should not be changed.
- 4.8 An abstract method has no meaningful definition. As a result, each derived class must redefine it to provide a definition. A base class that contains an abstract method is abstract and objects of the class cannot be defined.
- 4.9 An interface is a class that contains a protocol but no implementation. As such it consists exclusively of public abstract methods and public static final fields. This is different from abstract classes, which may have partial implementations.
- 4.10 The Java I/O library classes can be divided into two groups, input and output. The I/O is done via input and output streams. Byte oriented reading and writing is provided via `InputStream` and `OutputStream` classes whereas character-oriented I/O is provided via `Reader` and `Writer` classes. Most other classes for I/O (for files, pipes, sockets etc) are derived from these four classes.

```

Object
  FilterInputStream
    DataInputStream
    InflaterInputStream (java.util.zip)
    GZipInputStream (java.util.zip)
  FilterOutputStream
    DataOutputStream
    DeflaterOutputStream (java.util.zip)
    GZipOutputStream (java.util.zip)
  InputStream
    FileInputStream
    SocketInputStream (hidden)
    ObjectInputStream
  OutputStream
    FileOutputStream
    SocketOutputStream (hidden)
    ObjectOutputStream
  Reader
    BufferedReader
    InputStreamReader
    FileReader
  Writer
    PrintWriter

```

- 4.11 Generic algorithms are implemented by using `Object` or some other interface (such as `Comparable`) as parameters and return types. In other words, generic algorithms are implemented by using inheritance. To store primitive types in a generic container, wrapper classes must be used.
- 4.12 A wrapper pattern is used to store an existing entity (e.g., a class) and add operations that the original type does not support. An adapter pattern is used to change the interface of an existing class to conform to another interface. Unlike the wrapper pattern, the aim of the adapter pattern is not to change or add functionality.

- 4.13 One way to implement an adaptor is via composition wherein a new class is defined that creates an instance of the original class and redefines the interface. The other method is via inheritance. Inheritance adds new methods and may leave the original methods intact whereas via composition, we create a class that implements just the new interface. A function object is implemented as a class with no data object and one method.
- 4.14 A local class is a class that is placed inside a method. The local class is visible only inside the method. An anonymous class is a class with no name. It is often used to implement function objects (the syntax allows writing `new Interface()` with the implementation of `Interface` immediately following it).
- 4.15 Type erasure is the process of converting a generic class to a non-generic implementation. Due to type erasure, generic types cannot be primitive types. Similarly, `instanceof` tests cannot be used with generic types, nor can a generic type be instantiated or referenced in a static context. Arrays of generic types cannot be created.
- 4.16 In Java, arrays are covariant, however generic collections are not. Wildcards are used to express subclasses (or superclasses) of parameter types in generic collections. The type bound is specified inside the angle brackets `<>`, and it specifies properties that the parameter types must have.

IN THEORY

- 4.17 a. false b. true c. true d. true e. true f. true g. false h. true i. true j. true (see `java.io.Serializable`) k. true l. false m. false n. true o. true p. false q. true r. false s. true t. false u. true v. true
- 4.18 Objects of the `File`, `FileInputStream`, and `FileReader` are all acceptable parameters for a `Scanner` constructor. The latter two (`FileInputStream` and `FileReader`) can use the constructors that accept the `InputStream` and `Readable` parameters respectively.
- 4.19 Consider the case in which a method `foo()` contains a local class and returns an object of type `LocalClass`. In this case, if the local class accesses `x`, then it continues to be available after `foo` terminates.

```

Typetest foo ()
{
    final int x = 1;
    class LocalClass() implements Typetest
    {
        public int foo()
        { return x; }
    }
    return new localClass();
}

```

- 4.20 The output is 512 and the signature is `String getX()`. After changing as given in (c), the output is 17 and the signature is `int getX()`. If `Class1` is changed as in (d), we get an error as `Class2` is expecting a method `getX` returning an integer. If code inlining were allowed, the result would have been 17.
- 4.21 a) The last line throws a `ClassCastException` as the element in the array must be specifically cast, not the array itself. b) The last line contains an unnecessary cast. c) No errors. d) No errors.

IN PRACTICE

- 4.22 One solution is as follows:

```

public static <AnyType> void copy( AnyType [ ] arr1, AnyType [ ] arr2 )
{
    for ( AnyType val : arr1 )
        arr2[2] = val;
}

```

4.23 The class below includes the generic method `min` and a test routine. `max` is similar.

```

public class Ex0420
{
    public static Comparable min( Comparable a, Comparable b )
    {
        return a.lessThan( b ) ? a : b;
    }

    public static void main( String [ ] args )
    {
        int x = 5, y = 7;
        System.out.println(min(new MyInteger( x ), new MyInteger( y )));
    }
}

```

4.24 One solution is a follows:

```

public static Comparable min( Comparable [ ] a )
{
    int minIndex = 0;
    for ( int i = 1; i < a.length; i++ )
        if ( a[minIndex].compareTo(a[i]) > 0 )
            minIndex = i;
    return a[minIndex];
}

```

4.25 One solution is as follows:

```

class MaxTwo
{
    public static Comparable [ ] max2( Comparable [ ] a )
    {
        Comparable [ ] obj1 = new Comparable [2];
        int maxIndex0 = 0;
        int maxIndex1 = 0;
        if( a[0].compareTo(a[1]) > 0 )
            maxIndex0 = 1;
        else
            maxIndex1 = 1;

        for( int i = 1; i < a.length; i++ )
        {
            if( a[maxIndex0].compareTo(a[i]) > 0 )
            {
                maxIndex1 = maxIndex0;
                maxIndex0 = i;
            }
        }
        obj1[0] = a[maxIndex0];
        obj1[1] = a[maxIndex1];
    }
}

```

```

        return obj1;
    }

    public static void main( String [] args )
    {
        String [] st1 = { "B", "C", "A", "F" };
        Comparable [] st2 = new Comparable[2];
        st2 = max2(st1);
        System.out.println(st2[0] + " " + st2[1]);
    }
}

```

4.26 One possible solution is:

```

class SortWithMin
{
    public static int findmin( Comparable [] a, int start )
    {
        int minIndex = start;
        for( int i = start; i < a.length; i++ )
            if( a[minIndex].compareTo(a[i]) > 0 )
                minIndex = i;
        return minIndex;
    }

    public static void sort (Comparable [] a)
    {
        int tempIndex;
        Comparable temp;
        for (int i = 0; i < a.length; i++)
        {
            tempIndex = findmin(a,i);
            temp = a[i];
            a[i] = a[tempIndex];
            a[tempIndex] = temp;
        }
    }

    public static void main (String [] args)
    {
        String [] st1 = { "B", "C", "A", "F" };
        sort(st1);
        for( int i = 0; i < st1.length; i++ )
            System.out.println( st1[ i] );

        Shape [] a = new Shape[ ] { new Circle( 2.0 ), new Circle( 1.0),
                                     new Circle( 3.0 ) };
        sort(a);
        for( int i = 0; i < a.length; i++ )
            System.out.println( a[ i]. area() );
    }
}

```

This assumes that Shape implements Comparable.

4.27 One possible solution is:

```

public class Circle extends Shape
{
    public Circle( double rad )
    {
        if( rad >= 0 )
            radius = rad;
        else
            throw new InvalidArgumentException();
    }

    private double radius;
}

public class Rectangle extends Shape
{
    public Rectangle( double len, double wid )
    {
        if( (len >= 0) && (wid >= 0) )
        {
            length = len; width = wid;
        }
        else
            throw new InvalidArgumentException();
    }

    private double length;
    private double width;
}

public class Square extends Rectangle
{
    public Square( double side )
    {
        super( side, side );
    }

    public String toString( )
    {
        return "Square: " + getLength( );
    }
}

public class InvalidArgumentException extends RuntimeException
{
}

```

4.28 One possible implementation is:

```

public class Ellipse extends Shape implements Stretchable
{
    private double length;
    private double width;

    public Ellipse (double length, double width)
    {
        this.length = length;
        this.width = width;
    }
}

```

```

    }

    public double area()
    {
        return Math.PI * length/2.0 * width/2.0;
    }

    public double perimeter()
    {
        double asq = Math.pow(length/2.0, 2);
        double bsq = Math.pow(width/2.0, 2);
        return 2.0 * Math.PI * Math.sqrt((asq+bsq)/2.0);
    }

    public String toString()
    {
        return "Ellipse: " + length + " " + width;
    }

    public double getLength()
    {
        return length;
    }

    public double getWidth()
    {
        return width;
    }

    public void stretch(double factor)
    {
        if( factor <= 0 )
            throw new IllegalArgumentException( );

        if( length > width )
            length *= factor;
        else
            width *= factor;
    }
}

```

- 4.29 One possible solution follows. However note that since we are dealing with an underlying Object, we can only perform an equality comparison.

```

// MemoryCell class
// Object read( )      --> Returns the stored value
// void write( Object x ) --> x is stored

public class MemoryCell implements Comparable<MemoryCell>
{
    // Public methods
    public Object read( )
    {
        return storedValue;
    }

    public void write( Object x )

```



```

    {
        storedValue = x;
    }

    public int compareTo(MemoryCell obj)
    {
        if (this.storedValue.equals(obj.read()))
            return 0;
        else
            return -1;
    }

    // Private internal data representation
    private Object storedValue;
}

```

- 4.30 Since Shape already implements Comparable<Shape>, we can use Shape's compareTo method for the comparison in Circle (and not change the code). If we do attempt to modify Circle class as follows, we'll obtain a compile time error message (as follows). If Shape does not implement Comparable<Shape>, then the Circle implementation below is valid.

```

public class Circle extends Shape implements Comparable<Circle>
{
    public Circle( double rad )
    {
        radius = rad;
    }

    public double area( )
    {
        return Math.PI * radius * radius;
    }

    public double perimeter( )
    {
        return 2 * Math.PI * radius;
    }

    public String toString( )
    {
        return "Circle: " + radius;
    }

    public int compareTo(Circle other)
    {
        if (radius == other.radius)
            return 0;
        else if (radius > other.radius)
            return 1;
        else
            return -1;
    }

    private double radius;
}

```

```
./Circle.java:1: java.lang.Comparable cannot be inherited with
different arguments: <Circle> and <Shape>
public class Circle extends Shape implements Comparable<Circle>
```

4.31 A possible solution is:

```
package weiss.math;

import java.math.*;

public class BigRational4 implements Comparable<BigRational4>
{
    public static final BigRational4 ZERO = new BigRational4( );
    public static final BigRational4 ONE = new BigRational4( "1" );

    public BigRational4( String str )
    {
        if( str.length( ) == 0 )
            throw new IllegalArgumentException( "Zero-length string" );

        // Check for "/"
        int slashIndex = str.indexOf( '/' );
        if( slashIndex == -1 )
        {
            // no denominator... use 1
            den = BigInteger.ONE;
            num = new BigInteger( str.trim( ) );
        }
        else
        {
            den = new BigInteger( str.substring( slashIndex + 1 ).trim() );
            num = new BigInteger( str.substring( 0, slashIndex ).trim() );
            check00( ); fixSigns( ); reduce( );
        }
    }

    private void check00( )
    {
        if( num.equals( BigInteger.ZERO ) &&
            den.equals( BigInteger.ZERO ) )
            throw new IllegalArgumentException( "0/0" );
    }

    // Ensure that the denominator is positive
    private void fixSigns( )
    {
        if( den.compareTo( BigInteger.ZERO ) < 0 )
        {
            num = num.negate( );
            den = den.negate( );
        }
    }

    // Divide num and den by their gcd
    private void reduce( )
```

```

{
    BigInteger gcd = num.gcd( den );
    num = num.divide( gcd );
    den = den.divide( gcd );
}

public BigInteger4( BigInteger n, BigInteger d )
{
    num = n;
    den = d;
    check00( ); fixSigns( ); reduce( );
}

public BigInteger4( BigInteger n )
{
    this( n, BigInteger.ONE );
}

public BigInteger4( )
{
    this( BigInteger.ZERO );
}

public BigInteger4 abs( )
{
    return new BigInteger4( num.abs( ), den );
}

public BigInteger4 negate( )
{
    return new BigInteger4( num.negate( ), den );
}

public BigInteger4 add( BigInteger4 other )
{
    BigInteger newNumerator =
        num.multiply( other.den ).add(
            other.num.multiply( den ) );
    BigInteger newDenominator =
        den.multiply( other.den );

    return new BigInteger4( newNumerator, newDenominator );
}

public BigInteger4 subtract( BigInteger4 other )
{
    return add( other.negate( ) );
}

public BigInteger4 multiply( BigInteger4 other )
{
    BigInteger newNumerator = num.multiply( other.num );
    BigInteger newDenominator = den.multiply( other.den );

    return new BigInteger4( newNumerator, newDenominator );
}

```

```

public BigRational4 divide( BigRational4 other )
{
    if( other.num.equals( BigInteger.ZERO ) &&
        num.equals( BigInteger.ZERO ) )
        throw new ArithmeticException( "ZERO DIVIDE BY ZERO" );

    BigInteger newNumerator = num.multiply( other.den );
    BigInteger newDenom = den.multiply( other.num );

    return new BigRational4( newNumerator, newDenom );
}

public boolean equals( Object other )
{
    if( ! ( other instanceof BigRational4 ) )
        return false;

    BigRational4 rhs = (BigRational4) other;

    return num.equals( rhs.num ) && den.equals( rhs.den );
}

public String toString( )
{
    if( den.equals( BigInteger.ZERO ) )
        if( num.compareTo( BigInteger.ZERO ) < 0 )
            return "-infinity";
        else
            return "infinity";

    if( den.equals( BigInteger.ONE ) )
        return num.toString( );
    else
        return num + "/" + den;
}

private BigDecimal toBigDecimal()
{
    BigDecimal decnum = new BigDecimal( num, MathContext.UNLIMITED );
    BigDecimal result = null;

    try
    {
        result = decnum.divide( new BigDecimal( den,
            MathContext.UNLIMITED ), MathContext.UNLIMITED );
    }
    catch ( ArithmeticException ae )
    {
        if ( ae.getMessage().equals(
            "Non-terminating decimal expansion; " +
            "no exact representable decimal result." ) )
            result = decnum.divide(
                new BigDecimal( den, MathContext.UNLIMITED ),
                MathContext.DECIMAL128 );
    }
    return result;
}

```

```

    public int compareTo (BigRational4 other)
    {
        BigDecimal otherDec = other.toBigDecimal();
        BigDecimal thisDec = this.toBigDecimal();
        return thisDec.compareTo(otherDec);
    }

    private BigInteger num; // only this can be neg
    private BigInteger den;
}

```

4.32 A possible solution is:

```

import java.util.*;

public class Polynomial implements Comparable<Polynomial>
{
    private int degree = -1;
    private int[] coeff;

    public Polynomial ()
    {
        degree = 0;
        coeff = new int[1];
        coeff[0] = 0;
    }

    public Polynomial (Polynomial poly)
    {
        this.degree = poly.degree;
        coeff = new int[poly.coeff.length];
        for (int i=0; i<poly.coeff.length; i++)
            coeff[i] = poly.coeff[i];
    }

    public Polynomial (String str) throws InvalidSpecificationException
    {
        // input format is pairs of ints (e.g 2,3) separated by a comma
        // which
        // indicate the coefficient and the exponent (2x^3). Pairs are
        // separated by
        // whitespace. For example: 2,3 4,2 -5,0 translates to 2x^3 +
        // 4x^2 -5
        // The largest exponent must appear in the first term (the
        // degree). Other
        // terms can appear in any order, but if there are duplicates
        // terms with
        // the same power, only the last term is used. Thus, it's not
        // overly robust.
        Scanner poly = new Scanner(str);
        while (poly.hasNext())
        {
            int cof = 0;
            int exponent = 0;
            String term = poly.next();
            Scanner termScan = new Scanner(term);

```

```

        termScan.useDelimiter(",");

        // read each part of the term (coefficient,exponent)
        if (termScan.hasNextInt())
            cof = termScan.nextInt();
        else
            throw new InvalidSpecificationException("Invalid term: " +
term);
        if (termScan.hasNextInt())
            exponent = termScan.nextInt();
        else
            throw new InvalidSpecificationException("Invalid term: " +
term);

        // set data; degree of poly is largest exponent
        if (degree < exponent)
            degree = exponent;

        // create array and initialize if needed, then add new value
to array
        if (coeff == null)
        {
            coeff = new int[degree+1];
            for (int i = 0; i < coeff.length; i++)
                coeff[i] = 0;
        }
        coeff[exponent] = cof;
    }
}

// Duplicates the current polynomial and then negates each term.
public Polynomial negate()
{
    if (coeff == null)
        return null;

    Polynomial result = new Polynomial(this);
    for (int i = 0; i < result.coeff.length; i++)
        if (result.coeff[i] != 0)
            result.coeff[i] *= -1;

    return result;
}

// Adds the two polynomials together, modifying neither, and
returning
// a new resulting polynomial.
public Polynomial add(Polynomial rhs)
{
    // Determine which polynomial is larger, that's the one the
result
// will be based on, and at the end of the method, that's the one
we
// return.
    Polynomial result;
    Polynomial lhs;
    if (degree >= rhs.degree)

```

```

        {
            result = new Polynomial(this);
            lhs = rhs;
        } else
        {
            result = new Polynomial(rhs);
            lhs = this;
        }

        for (int i = lhs.degree; i >= 0; i--)
            result.coeff[i] += lhs.coeff[i];
        return result;
    }

    // Subtracts the rhs polynomial from the lhs polynomial, modifying
    // neither,
    // and returning a new resulting polynomial.
    public Polynomial subtract(Polynomial rhs)
    {
        return this.add(rhs.negate());
    }

    public Polynomial multiply(Polynomial rhs)
    {
        Polynomial result = new Polynomial();
        result.degree = this.degree + rhs.degree;
        result.coeff = new int[result.degree+1];

        for (int i = 0; i < rhs.coeff.length; i++)
            for (int j = 0; j < this.coeff.length; j++)
                result.coeff[i+j] += this.coeff[j] * rhs.coeff[i];
        return result;
    }

    // Returns a true value if the lhs and rhs polynomials are equal in
    // all
    // respects.
    public boolean equals(Object rhs)
    {
        Polynomial prhs = (Polynomial) rhs;
        boolean result = true;
        if (degree != prhs.degree)
            result = false;

        for (int i = 0; !result && i < degree; i++)
            if (coeff[i] != prhs.coeff[i])
                result = false;
        return result;
    }

    // Returns a string representation of the current polynomial.
    public String toString()
    {
        String result = "";
        for (int i = degree; i >= 0; i--)
        {
            if (i != degree && coeff[i] > 0)

```

```

        result += '+';

        if (coeff[i] != 0)
            result += coeff[i];
        if (i > 1 && coeff[i] != 0)
            result += "x^" + i + ' ';
        else if (i == 1 && coeff[i] != 0)
            result += "x" + ' ';
    }
    return result;
}

public int compareTo(Polynomial rhs)
{
    if (degree < rhs.degree)
        return -1;
    else if (degree == rhs.degree)
        return 0;
    else
        return 1;
}

class InvalidSpecificationException extends IllegalArgumentException
{
    public InvalidSpecificationException(String msg)
    {
        super (msg);
    }
}
}

```

- 4.33 Assuming the Shape class does not implement Comparable<Shape> (see problem 4.30), one solution would be:

```

public class Square extends Shape implements Comparable<Square>
{
    public Square (double side)
    {
        this.side = side;
    }

    public double area( )
    {
        return Math.pow(side, 2.0);
    }

    public double perimeter( )
    {
        return side * 4.0;
    }

    public int compareTo(Square other)
    {
        if (side == other.side)
            return 0;
        else if (side < other.side)
            return -1;
    }
}

```



```

        else
            return 1;
    }

    private double side;
}

```

4.34 A possible solution (that assumes a right triangle) is:

```

public class Triangle extends Shape implements Stretchable
{
    // Assumes a right triangle
    public Triangle (double base, double height)
    {
        this.base = base;
        this.height = height;
    }

    public double area( )
    {
        return 0.5 * base * height;
    }

    public double perimeter( )
    {
        return base + height + (base*base) + (height*height);
    }

    public void stretch (double factor)
    {
        base = base * factor;
        height = height * factor;
    }

    public String toString ( )
    {
        return "Triangle base: " + base + " height: " + height;
    }

    private double base;
    private double height;
}

```

4.35 A solution is:

```

public static void stretchAll( ArrayList<? extends Stretchable> arr,
    double factor )
{
    for( Stretchable s : arr )
        s.stretch( factor );
}

```

4.36 A possible solution is:

```

class Person implements Comparable
{
    public int compareTo(Object rhs)

```

```

    {
        Person p2 = (Person) rhs;
        return (name.compareTo(p2.name));
    }
    // Class is identical, with changes in bold.
}

```

4.37 A solution is:

```

public class SingleBuffer <AnyType> {
    private AnyType theItem;
    private boolean isEmpty;

    public SingleBuffer( )
    {
        theItem = null;
        isEmpty = true;
    }

    public SingleBuffer( AnyType item )
    {
        theItem = item;
        if ( theItem != null )
            isEmpty = false;
        else
            isEmpty = true;
    }

    public AnyType get( ) throws SingleBufferException
    {
        if ( isEmpty )
            throw new SingleBufferException( "Buffer is empty!" );

        isEmpty = true;
        AnyType item = theItem;
        theItem = null;
        return item;
    }

    public void put( AnyType item ) throws SingleBufferException
    {
        if ( !isEmpty )
            throw new SingleBufferException(
                "Buffer must be emptied by insertion!" );
        theItem = item;
        isEmpty = false;
    }
}

public class SingleBufferException extends Exception
{
    public SingleBufferException( String msg )
    {
        super( msg );
    }
}

```

- 4.38 Here is an even fancier version that uses iterators, inner classes, nested classes, and comparators. Of course it has way too many forward references for use at this point in the course, but you can remove complications as needed.

```
import java.util.*;
/**
 * A SortedArrayList stores objects in sorted order.
 * The SortedArrayList supports the add operation.
 * size is also provided, and so is a method
 * that returns an Iterator.
 * Of course a get routine could be provided too
 * as could numerous other routines..
 *
 * This example illustrates both instance inner classes
 * and static inner classes.
 * An instance inner class is used in the typical iterator pattern.
 * A static inner class is used to define a default comparator.
 */
class SortedArrayList
{
    // The list, in sorted order
    private ArrayList data = new ArrayList( );

    // The comparator object
    private Comparator cmp;

    /**
     * Construct the SortedArrayList with specified Comparator.
     * @param compare The Comparator object.
     */
    public SortedArrayList( Comparator compare )
    {
        cmp = compare;
    }

    /**
     * Construct the SortedArrayList using natural ordering
     * If objects are not Comparable, an exception will be
     * thrown during an add operation.
     */
    public SortedArrayList( )
    {
        this( new DefaultComparator( ) );
    }

    private static class DefaultComparator implements Comparator
    {
        public int compare( Object obj1, Object obj2 )
        {
            return ((Comparable) obj1).compareTo( obj2 );
        }
    }

    /**
     * Add a new value to this SortedArrayList, maintaining sorted
     * order.
     * @param x The Object to add.
     */
}
```

```

    */
    public void add( Object x )
    {
        data.add( x ); // add at the end for now
        int i = data.size( ) - 1;

        // Slide x over to correct position
        for( ; i > 0 && cmp.compare( data.get( i - 1 ), x ) > 0; i-- )
            data.set( i, data.get( i - 1 ) );
        data.set( i, x );
    }

    /**
     * Return the number of items in this SortedArrayList.
     * @return the number of items in this SortedArrayList.
     */
    public int size( )
    {
        return data.size( );
    }

    /**
     * Return an Iterator that can be used to traverse
     * this SortedArrayList. The remove operation is unimplemented.
     * @return An Iterator that can be used to traverse this
     * SortedArrayList.
     */
    public Iterator iterator( )
    {
        return new SortedIterator( );
    }

    private class SortedIterator implements Iterator
    {
        private int current = 0;

        public boolean hasNext( )
        {
            return current < size( );
        }

        public Object next( )
        {
            return data.get( current++ );
        }

        public void remove( )
        {
            throw new UnsupportedOperationException( );
        }
    }
}

class TestSortedArrayList
{
    public static String listToString( SortedArrayList list )
    {

```

```

        Iterator itr = list.iterator( );
        StringBuffer sb = new StringBuffer( );
        for( int i = 0; itr.hasNext( ); i++ )
            sb.append( "[" + i + "]" + itr.next( ) + " " );
        return new String( sb );
    }

    // Test by inserting 20 Strings
    public static void main( String[] args )
    {
        SortedArrayList list = new SortedArrayList( );
        for( int i = 0; i < 20; i++ )
            list.add( "" + i );
        System.out.println( "Using iterator" );
        System.out.println( listToString( list ) );
    }
}

```

4.39 One solution is as follows:

```

import weiss.util.Comparator;

public class StringComparator implements Comparator<String>
{
    public int compare (String str1, String str2)
    {
        return str1.compareToIgnoreCase(str2);
    }
}

```

4.40 Code solutions to all parts are as follows:

```

package weiss.util;

/**
 * ContentsChecker function object interface.
 */
public interface ContentsChecker
{
    /**
     * Returns a true value if data matches a criteria (prime, etc.)
     */
    public boolean check(int [] data);
}

```

```

// Accepts an array of ints and a ContentsChecker object. The array is
// then searched to determine if there are any ints contained in the
// array that match the criteria of the ContentsChecker.
public static boolean contains (int [] input, ContentsChecker cmp)
{
    return cmp.check(input);
}

```

```

import weiss.util.ContentsChecker;

public class Negative implements ContentsChecker
    // Checks to determine if any of the items of the data array are
    // negative.
    public boolean check (int [] data)
    {
        boolean negative = false;
        for (int index = 0; !negative && index < data.length; index++)
        {
            if (data[index] < 0)
                negative = true;
        }
        return negative;
    }
}

```

```

import weiss.util.ContentsChecker;

public class Prime implements ContentsChecker
{
    // Checks to determine if any of the items of the data array are
    // prime values. This algorithm uses the naive algorithm (see
    // http://en.wikipedia.org/wiki/Primality\_test#Na.C3.AFve\_methods)
    // to detect primality.
    public boolean check (int [] data)
    {
        boolean prime = false;
        for (int index = 0; index < data.length; index++)
        {
            int sqrtNum = (new Double(Math.sqrt(data[index]))).intValue();
            boolean thisTest = true;
            for (int tester = 2; thisTest && tester <= sqrtNum; tester++)
                if (data[index] % tester == 0)
                    thisTest = false;
            if (thisTest == true)
                prime = true;
        }
        return prime;
    }
}

```

```

import weiss.util.ContentsChecker;

public class PerfectSquare implements ContentsChecker
{
    // Checks to determine if any of the items of the data array are
    // perfect squares.
    public boolean check (int [] data)
    {
        boolean perfectSquare = false;
        for (int index = 0; !perfectSquare && index < data.length;

```

```

        index++)
        {
            if (Math.pow(Math rint(Math.sqrt(data[index])), 2.0) ==
                data[index])
                perfectSquare = true;
        }
        return perfectSquare;
    }
}

```

4.41 Code solutions to all parts are as follows:

```

package weiss.util;

/**
 * ArrayComputation function object interface.
 */
public interface ArrayTransformation
{
    /**
     * Performs an operation on an array
     */
    public void transform (double [] input, double[] transform);
}

```

```

// Performs a specified transformation on an input array, placing the
// output into the specified output array. An IllegalArgumentException
// is thrown if the array lengths are not equal.
public static void transform (double[] input, double[] output,
    ArrayTransformation trans) throws IllegalArgumentException
{
    if (input.length != output.length)
        throw new IllegalArgumentException();

    trans.transform(input, output);
}

```

```

import weiss.util.ArrayTransformation;

public class ComputeSquare implements ArrayTransformation
{
    /**
     * Computes the square of each member of the input array.
     * The results are returned in the new array.
     */
    public void transform (double [] input, double [] output)
    {
        for (int index = 0; index < input.length; index++)
            output[index] = Math.pow(input[index], 2.0);
    }
}

```

```

import weiss.util.ArrayTransformation;

```

```

public class ComputeAbsoluteValue implements ArrayTransformation
{
    /**
     * Computes the absolute value of each member of the input array.
     * The results are returned in the new array.
     */
    public void transform (double [] input, double [] output)
    {
        for (int index = 0; index < input.length; index++)
            output[index] = Math.abs(input[index]);
    }
}

```

4.42 A solution is as follows:

```

package weiss.util;

/**
 * ContentsChecker2 function object interface.
 */
public interface ContentsChecker2
{
    /**
     * Returns a true value if data matches a criteria (prime, etc.)
     */
    public <AnyType> boolean check(AnyType[] data);
}

```

```

// Accepts any array and a ContentsChecker2 object. The array is
// then searched to determine if there are any items contained in
// the array that match the criteria of the ContentsChecker2.
public static <AnyType> boolean contains2 (AnyType[] input,
    ContentsChecker2 cmp)
{
    return cmp.check(input);
}

```

```

import weiss.util.ContentsChecker2;

public class Negative2 implements ContentsChecker2
{
    // Checks to determine if any of the items of the
    // data array are negative.
    public <AnyType> boolean check (AnyType[] data)
    {
        boolean negative = false;
        for (int index = 0; !negative && index <
            data.length; index++)
        {
            Integer zero = new Integer(0);
            if (zero.compareTo((Integer) data[index]) > 1)
                negative = true;
        }
    }
}

```



```

        return negative;
    }
}

```

```

import weiss.util.ContentsChecker2;

public class Prime2 implements ContentsChecker2
{
    // Checks to determine if any of the items of the
    // data array are prime values. This algorithm uses
    // the naive algorithm (see
    // http://en.wikipedia.org/wiki/Primality\_test#Na.C3.AFve\_methods)
    // to detect primality.
    public <AnyType> boolean check (AnyType[] data)
    {
        boolean prime = false;
        for (int index = 0; index < data.length; index++)
        {
            int sqrtNum = (new Double(Math.sqrt(
                (Integer) data[index]))).intValue();
            boolean thisTest = true;
            for (int tester = 2; thisTest && tester <= sqrtNum;
                tester++)
            {
                if ((Integer) data[index] % tester == 0)
                    thisTest = false;
            }
            if (thisTest == true)
                prime = true;
        }
        return prime;
    }
}

```

```

import weiss.util.ContentsChecker2;

public class PerfectSquare2 implements ContentsChecker2
{
    // Checks to determine if any of the items of the data
    // array are perfect squares.
    public <AnyType> boolean check (AnyType[] data)
    {
        boolean perfectSquare = false;
        for (int index = 0; !perfectSquare && index < data.length;
            index++)
        {
            double valueSqr = Math.pow(Math rint(Math.sqrt(
                (Integer) data[index])), 2.0);
            if (valueSqr == (Integer) data[index])
                perfectSquare = true;
        }
        return perfectSquare;
    }
}

```

4.43 A solution is as follows:

```

package weiss.util;

/**
 * ArrayComputation function object interface.
 */
public interface ArrayTransformation2
{
    /**
     * Performs an operation on an array
     */
    public <AnyType> void transform (AnyType[] input,
                                   AnyType[] transform);
}

```

```

// Performs a specified transformation on an input array,
// placing the output into the specified output array. An
// IllegalArgumentException is thrown if the array lengths
// are not equal.
public static <AnyType> void transform2 (AnyType[] input,
                                         AnyType[] output, ArrayTransformation2 trans) throws
                                         IllegalArgumentException
{
    if (input.length != output.length)
        throw new IllegalArgumentException();

    trans.transform(input, output);
}

```

```

import weiss.util.ArrayTransformation2;

public class ComputeSquare2 implements ArrayTransformation2
{
    /**
     * Computes the square of each member of the input array.
     * The results are returned in the new array.
     */
    public <AnyType> void transform (AnyType[] input, AnyType[] output)
    {
        for (int index = 0; index < input.length; index++)
        {
            Double newDbl = new Double((Math.pow(
                (Double) input[index], 2.0)));
            output[index] = (AnyType) newDbl;
        }
    }
}

import weiss.util.ArrayTransformation2;

public class ComputeAbsoluteValue2 implements ArrayTransformation2
{
    /**
     * Computes the absolute value of each member of the input array.
     * The results are returned in the new array.
     */
}

```

```

public <AnyType> void transform (AnyType[] input, AnyType[] output)
{
    for (int index = 0; index < input.length; index++)
    {
        Double newDb1 = new Double(Math.abs((Double) input[index]));
        output[index] = (AnyType) newDb1;
    }
}

```

4.44 One solution is as follows:

```

public interface Matchable
{
    boolean matches(int a);
}

class EqualsZero implements Matchable
{
    public boolean matches(int a)
    {
        return a == 0;
    }
}

class CountTester
{
    public static int countMatches( int [] a, Matchable func )
    {
        int num = 0;
        for( int i = 0; i < a.length; i++ )
            if( func.matches( a[i]) )
                num++;
        return num;
    }

    public static void main (String [] args)
    {
        int [] a = { 4, 0, 5, 0, 0};
        System.out.println(countMatches(a, new EqualsZero()));
    }
}

```

4.45 One solution is as follows:

```

class EqualsK implements Matchable
{
    private int k;
    public EqualsK( int initialk )
    {
        k = initialk;
    }

    public EqualsK()
    {
        this( 0 );
    }
}

```

```

    public boolean matches(int a)
    {
        return a == k;
    }
}

class Tester
{
    public static void main (String [] args)
    {
        int [] a = { 4, 0, 5, 0, 0};
        System.out.println( CountTester.countMatches(a, new EqualsK(5)));
    }
}

```

4.3 Exam Questions

- 4.1 Which term describes the ability of a reference type to reference objects of several different types?
- composition
 - encapsulation
 - information hiding
 - polymorphism
 - static binding
- 4.2 In the derived class, which data members of the base class are visible?
- those in the public section only
 - those in the protected section only
 - those in the public or protected sections
 - those in the public or private sections
 - all data members are visible
- 4.3 For which methods is dynamic binding used?
- all class methods
 - all static methods
 - final methods
 - non-static, non-final class methods
 - none of the above
- 4.4 Which of the following is true about an abstract base class?
- no constructors should be provided
 - at least one member function should be abstract
 - no objects of the class can be created
 - exactly two of the above
 - all of (a), (b), and (c)
- 4.5 When is a method abstract?
- When it is constant throughout the inheritance hierarchy
 - Its definition changes in the inheritance hierarchy, but there is a reasonable default
 - No reasonable default can be provided, and it must be defined in the inheritance hierarchy
 - Always
 - none of the above
- 4.6 Which of the following are not allowed in an interface?

- a. public methods
 - b. static methods
 - c. final methods
 - d. public fields
 - e. static fields
- 4.7 How is generic programming implemented in Java 1.5?
- a. by inheritance
 - b. import directive
 - c. package statement
 - d. private methods
 - e. using a template
- 4.8 Which clause is used to derive a new class?
- a. extends
 - b. implements
 - c. import
 - d. throw
 - e. at least two of the above
- 4.9 Which of the following is false?
- a. An abstract class may implement an interface.
 - b. A class that contains an abstract method must be declared abstract itself or a compiler message will result.
 - c. A class may implement multiple interfaces.
 - d. A class that extends another class may implement only one interface.
 - e. An interface may extend another interface.
- 4.10 Which of (a) - (c) is false?
- a. An adapter class is used to change the interface of an existing class.
 - b. A wrapper class stores a primitive types and add operations that primitive type does not support.
 - c. An adapter is used to provide a simpler interface with fewer methods
 - d. Both (a) and (c) are false
 - e. all of the above are false
- 4.11 Which of (a) - (d) is false?
- a. A nested class is a member of its outer class.
 - b. A nested class must be declared static.
 - c. Methods of a local class can access local variables of its outer class.
 - d. A local class is visible only inside the method in which it is declared.
 - e. Both (b) and (c) are false

Answers to Exam Questions

- 1. D
- 2. C
- 3. D
- 4. D
- 5. C
- 6. C
- 7. A
- 8. E
- 9. D
- 10. B
- 11. C