7. Final, static variables; abstract methods.

8. This should cause a compile time error – the compiler should not create Java bytecode for the `SampleClass`. Error messages might vary across compilers. On my system I received two error messages, both very much to the point:

```
SampleClass is not abstract and does not override abstract method
sampleMethod() in SampleInterface

sampleMethod() in SampleClass cannot implement sampleMethod() in
SampleInterface; attempting to use incompatible return type
```

9.

a. False Interfaces are not instantiated.
b. True An interface specifies some implementation requirements for a class.
c. False Only interfaces extend interfaces.
d. True Just so it also includes all of the methods listed in the interface.
e. False It must include methods for each method listed in the interface - they can be abstract methods however.
f. False You can only instantiate objects of a concrete class.
g. False Interface can only include abstract methods.

14.

```
//---------------------------------------------------------------------
// IntLogInterface.java       by Dale/Joyce/Weems              Chapter 2
//
// Interface for a class that implements a log of ints.
// A log "remembers" the elements placed into it.
//
// A log must have a "name".
//---------------------------------------------------------------------

package answers.ch02.intLogs;

public interface IntLogInterface
{
  void insert(int element);
  // Precondition:   This IntLog is not full.
  //
  // Places element into this IntLog.

  boolean isFull();
  // Returns true if this IntLog is full, otherwise returns false.

  int size();
  // Returns the number of ints in this IntLog.

  boolean contains(int element);
  // Returns true if element is in this IntLog,
  // otherwise returns false.

  void clear();
  // Makes this IntLog empty.

  String getName();
  // Returns the name of this IntLog.
```

```
  String toString();
  // Returns a nicely formatted string representing this IntLog.
}
```

15. You cannot instantiate an interface like you instantiate a class to create an object. Interfaces are typically used to define requirements for a class. You can define a class that implements an interface and then instantiate objects of that class.

18.

a. No ramifications.
b. The first time `insert` is called the object would attempt to access the `log` array using an illegal index of -1 and an exception would be thrown.
c. The `isFull` method will incorrectly return the value false when the `log` array is full.
d. The search for the parameter `element` will begin at the 2nd location in the array (index 1) so if a match is stored in the 1st location (index 0) the method will incorrectly report that it was not found (unless an "identical" match is stored elsewhere in the array).
e. Similar to answer d, except now it is the last of the used array locations that will not be considered.

21.

```
  public boolean isEmpty()
  // Returns true if this StringLog is empty, otherwise returns false.
  {
    if (lastIndex == (- 1))
      return true;
    else
      return false;
  }
```

22.

```
  public int howMany(String element)
  // Returns how many times element occurs in this StringLog.
  {
    int count = 0;
   for (int location = 0; location <= lastIndex; location++)
      if (element.equalsIgnoreCase(log[location]))  // if they match
        count++;
    return count;
  }
```

23.

```
  public boolean uniqInsert(String element)
  // Places element into this StringLog unless an
  // identical string (case insensitive match)
  // is already in this StringLog.
  //
  // Assumes StringLog not full if insertion required.
  {
    if (this.contains(element))
      return false;
    else
    {
      this.insert(element);
      return true;
```

```
    }
  }
```

24. You must be careful when deleting from the `log` array – for the `StringLog` to continue to work correctly you must fill in the contents of the array slot targeted for deletion. The algorithm used here is similar to that developed for removal of an element from an unsorted list in Chapter 6. We simply copy the element sitting in the `lastIndex` slot into the slot identified for deletion.

```
public boolean delete(String element)
// Deletes one occurence of element from this StringLog,
// if possible, and returns true. Otherwise returns false.
{
  boolean moreToSearch;
  int location = 0;
  boolean found = false;
  moreToSearch = (location <= lastIndex);

  while (moreToSearch && !found)
  {
    if (element.equalsIgnoreCase(log[location]))
      found = true;
    else
    {
      location++;
      moreToSearch = (location <= lastIndex);
    }
  }

  if (found)
  {
    log[location] = log[lastIndex];
    log[lastIndex] = null;
    lastIndex--;
  }
  return found;
}
```

25. The required method can, of course, be devised "from scratch". Our solution makes use of the method developed to answer Exercise 24.

```
public int deleteAll(String element)
// Deletes all occurences of element from this StringLog,
// if any, and returns the number of deletions.
{
  int count = 0;
  while (this.delete(element))
    count++;
  return count;
}
```

26. The assumption that the `StringLog` isn't empty makes this problem easier to solve. We simply set the smallest to the string in the first array slot and then use a for loop to compare the "smallest so far" to each subsequent element .. changing the "smallest so far" whenever we encounter a smaller element.

```
public String smallest()
// Precondition: this log contains at least one string
```

```
  //
  // Returns smallest element in this StringLog
  {
    String small = log[0];
    for (int i = 1; i <= lastIndex; i++)
      if (small.compareTo(log[i]) > 0)
        small = log[i];

    return small;
  }
```

28.

a.

```
//--------------------------------------------------------------------
// ArrayIntLog.java        by Dale/Joyce/Weems              Chapter 2
//
// Implements IntLogInterface using an array to hold the strings.
//--------------------------------------------------------------------

package answers.ch02.intLogs;

public class ArrayIntLog implements IntLogInterface
{
  protected String name;              // name of this IntLog
  protected int[] log;                // array that holds ints
  protected int lastIndex = -1;       // index of last string in array

  public ArrayIntLog(String name, int maxSize)
  // Precondition:   maxSize > 0
  //
  // Instantiates and returns a reference to an empty IntLog object
  // with name "name" and room for maxSize strings.
  {
    log = new int[maxSize];
    this.name = name;
  }

  public ArrayIntLog(String name)
  // Instantiates and returns a reference to an empty IntLog object
  // with name "name" and room for 100 strings.
  {
    log = new int[100];
    this.name = name;
  }

  public void insert(int element)
  // Precondition:   This IntLog is not full.
  //
  // Places element into this IntLog.
  {
    lastIndex++;
    log[lastIndex] = element;
  }

  public boolean isFull()
  // Returns true if this IntLog is full, otherwise returns false.
```

```
{
  if (lastIndex == (log.length - 1))
    return true;
  else
    return false;
}

public int size()
// Returns the number of ints in this IntLog.
{
  return (lastIndex + 1);
}

public boolean contains(int element)
// Returns true if element is in this IntLog,
// otherwise returns false.
{
  boolean moreToSearch;
  int location = 0;
  boolean found = false;
  moreToSearch = (location <= lastIndex);

  while (moreToSearch && !found)
  {
    if (element == log[location])  // if they match
      found = true;
    else
    {
      location++;
      moreToSearch = (location <= lastIndex);
    }
  }

 return found;
}

public void clear()
// Makes this IntLog empty.
{
  lastIndex = -1;
}

public String getName()
// Returns the name of this IntLog.
{
  return name;
}

public String toString()
// Returns a nicely formatted string representing this IntLog.
{
  String logString = "Log: " + name + "\n\n";

  for (int i = 0; i <= lastIndex; i++)
    logString = logString + (i+1) + ". " + log[i] + "\n";

  return logString;
```

```
    }
}
```

b.

```
//-------------------------------------------------------------------------
// TestLuck.java          by Dale/Joyce/Weems                  Chapter 2
//
// Solution to Exercise 28, Chapter 2
//-------------------------------------------------------------------------

import answers.ch02.intLogs.*;
import java.util.Random;

public class TestLuck
{
  public static void main(String[] args)
  {
    ArrayIntLog log = new ArrayIntLog("Test 26", 10000);

    Random rand = new Random();  // to generate random numbers

    int count = 0;
    int randNum = rand.nextInt(10000) + 1;

    while (!log.contains(randNum))
    {
      log.insert(randNum);
      count++;
      randNum = rand.nextInt(10000) + 1;
    }
    System.out.println(count);
  }
}
```

c. Answers will vary. When we ran our program ten times we got the following results: 93, 65, 151, 144, 166, 121, 170, 60, 201, 99. That's an average of 127. Most people would expect a higher average – in fact many people might expect that the average would be 5,000, i.e., half the random number range. However, you must remember that we are not simply choosing two random numbers and comparing them. Our problem is related to a famous, well-studied statistical problem called, the birthday problem (students can find information about the birthday problem on the web). Our surprising result of 127 is a reasonable result when evaluated with approximation formulas associated with the "general" birthday problem.

33. The body of the while loop is not in braces.
    The comment includes the call to increment the count variable.