Practice Test 2

COMPUTER SCIENCE A
SECTION I

Time—1 hour and 30 minutes
40 Questions

DIRECTIONS: Determine the answer to each of the following questions or incomplete statements, using the available space for any necessary scratchwork. Then decide which is the best of the choices given and fill in the corresponding oval on the answer sheet. Do not spend too much time on any one problem. (Note that the actual exam will feature an answer sheet, but you can practice on a piece of paper.)

NOTES:

- Assume that the classes in the Quick Reference have been imported where needed.
- Assume that variables and methods are declared within the context of an enclosing class.
- Assume that method calls that have no object or class name prefixed, and that are not shown within a complete class definition, appear within the context of an enclosing class.
- Assume that parameters in method calls are not `null` unless otherwise stated.

1. What output is produced by the following line of code?

```
System.out.println("\"This is\n very strange\"");
```

(A) `\This is\n very strange\`

(B) `"This is very strange"`

(C)
```
This is
very strange
```

(D)
```
\"This is
 very strange\"
```

(E)
```
"This is
 very strange"
```

2. A certain class, `SomeClass`, contains a method with the following header.

```
public int getValue(int n)
```

Suppose that methods with the following headers are now added to `SomeClass`.

I. `public int getValue()`

II. `public double getValue(int n)`

III. `public int getValue(double n)`

Which of the above headers will cause an error?

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I and III only

3. Consider the following statement.

```
int num = /* expression */;
```

Which of the following replacements for `/* expression */` creates in `num` a random integer from 2 to 50, including 2 and 50?

(A) `(int)(Math.random() * 50) - 2`
(B) `(int)(Math.random() * 49) - 2`

(C) `(int)(Math.random() * 49) + 2`

(D) `(int)(Math.random() * 50) + 2`

(E) `(int)(Math.random() * 48) + 2`

4. Consider the following code segment.

```
int num = 0, score = 10;
if (num != 0 && score / num > SOME_CONSTANT)
    statement1;
else
    statement2;
```

What is the result of executing this statement?

(A) An `ArithmeticException` will be thrown.

(B) A syntax error will occur.

(C) ***statement1***, but not ***statement2***, will be executed.

(D) ***statement2***, but not ***statement1***, will be executed.

(E) Neither ***statement1*** nor ***statement2*** will be executed; control will pass to the first statement following the `if` statement.

5. The following shuffle algorithm is used to shuffle an array of `int` values, `nums`.

```
public void shuffle ()
{
    for (int k = nums.length - 1; k > 0; k--)
    {
        int randPos = (int) (Math.random() * (k + 1));
        int temp = nums[k];
        nums[k] = nums[randPos];
        nums[randPos] = temp;
    }
}
```

Suppose the initial state of `nums` is `8, 7, 6, 5, 4`, and when the method is executed the values generated for `randPos` are `3, 2, 0, 0`, in that order. What element will be contained in `nums[2]` after execution?

(A) `8`

(B) `7`

(C) `6`

(D) `5`

(E) `4`

6. Consider the following instance variables and method `assignValues` in the same class.

```
private int numRows;
private int numCols;
private int[][] mat;

/** arr has numCols elements */
private void assignValues(int[] arr, int value)
{
    for (int k = 0; k < arr.length; k++)
        arr[k] = value;
}
```

Which of the following code segments will correctly assign `mat` to have the value `100` in each slot? You may assume that the instance variables have all been correctly initialized.

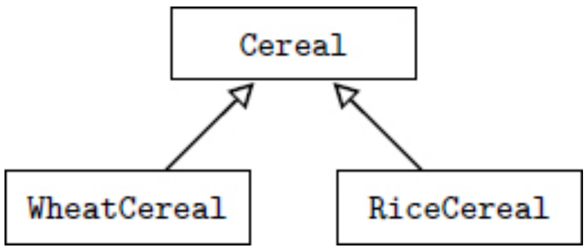
I.
```
for (int row = 0; row < numRows; row++)
    assignValues(mat[row], 100);
```

II.
```
for (int col = 0; col < numCols; col++)
    assignValues(mat[col], 100);
```

III.
```
for (int[] row: mat)
    for (int num: row)
        num = 100;
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I and III only

7. Consider the following inheritance hierarchy.



Which of the following declarations will not cause an error? You may assume that each of the classes above has a no-argument constructor.

I. `WheatCereal w = new Cereal();`

II. `Cereal c1 = new Cereal();`

III. `Cereal c2 = new RiceCereal();`

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I, II, and III

Questions 8 and 9 refer to the following class definitions.

```
public Class1
{
    public void method1()
    { /* implementation not shown */ }
}

public class Class2 extends Class1
{
    public void method2()
    { /* implementation not shown */ }

    //Private instance variables and other methods are not shown.
}

public class Class3 extends Class2
{
    public void method3(Class3 other)
    { /* implementation not shown */ }

    //Private instance variables and other methods are not shown.
}
```

8. Assuming that `Class1`, `Class2`, and `Class3` have no-argument constructors, which is (are) valid in a client class?

   I. `Class1 c1 = new Class2();`

   II. `Class2 c2 = new Class3();`

   III. `Class1 c3 = new Class3();`

   (A) I only
   (B) II only
   (C) III only
   (D) I and II only
   (E) I, II, and III

9. Consider the following declarations in a client class.

```
Class3 ob3 = new Class3();
Class2 ob2 = new Class2();
```

Which method calls would be legal?

I. `ob3.method1();`

II. `ob2.method3(ob3);`

III. `ob3.method3(ob2);`

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I, II, and III

10. Refer to the following program segment.

```
for (int n = 50; n > 0; n = n / 2)
    System.out.println(n);
```

How many lines of output will this segment produce?

(A) 50
(B) 49
(C) 7
(D) 6
(E) 5

11. Let `list` be an `ArrayList<String>` containing only these elements.

```
"John", "Mary", "Harry", "Luis"
```

Which of the following statements will cause an error to occur?

I. `list.set(2, "6");`

II. `list.add(4, "Pat");`

III. `String s = list.get(4);`

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I, II, and III

12. Consider the following static method.

```
public static int compute(int n)
{
    for (int i = 1; i < 4; i++)
        n *= n;
    return n;
}
```

Which of the following could replace the body of `compute`, so that the new version returns the identical result as the original for all `n`?

(A) `return 4 * n;`
(B) `return 8 * n;`
(C) `return 64 * n;`
(D) `return (int) Math.pow(n, 4);`
(E) `return (int) Math.pow(n, 8);`

13. Consider the following instance variable and method.

```
private int[] nums;

/** Precondition:  nums contains int values in no particular order.
 */
public int getValue()
{
    for (int k = 0; k < nums.length; k++)
    {
        if (nums[k] % 2 != 0)
            return k;
    }
    return -1;
}
```

Suppose the following statement is executed.

```
int j = getValue();
```

If the value returned in `j` is a positive integer, which of the following best describes the contents of `nums`?

(A) The only odd `int` in `nums` is at position `j`.
(B) All values in positions `0` through `j-1` are odd.
(C) All values in positions `0` through `j-1` are even.
(D) All values in positions `nums.length-1` down to `j+1` are odd.
(E) All values in positions `nums.length-1` down to `j+1` are even.

14. Consider the following method.

```
public int mystery (int n)
{
    if (n == 0)
        return 0;
    else if (n % 2 == 1)
        return n;
    else
        return n + mystery(n - 1);
}
```

What will be returned by a call to `mystery(6)?`

(A) 6

(B) 11

(C) 12

(D) 27

(E) 30

15. Consider the following code segment.

```
int num1 = value1, num2 = value2, num3 = value3;
while (num1 > num2 || num1 > num3)
{
    /* body of loop */
}
```

You may assume that `value1`, `value2`, and `value3` are `int` values. Which of the following is sufficient to guarantee that /∗ ***body of loop*** ∗/ will never be executed?

(A) There is no statement in /∗ ***body of loop*** ∗/ that leads to termination.

(B) `num1 < num2`

(C) `num1 < num3`

(D) `num1 > num2 && num1 > num3`

(E) `num1 < num2 && num1 < num3`

16. Consider the following two classes.

```
public class Performer
{
    public void act()
    {
        System.out.print(" bow");
        perform();
    }

    public void perform()
    {
        System.out.print(" act");
    }
}

public class Singer extends Performer
{
    public void act()
    {
        System.out.print(" rise");
        super.act();
        System.out.print(" encore");
    }

    public void perform()
    {
        System.out.print(" aria");
    }
}
```

Suppose the following declaration appears in a class other than `Performer` or `Singer`.

```
Performer p = new Singer();
```

What is printed as a result of the call `p.act();`?

(A) `rise bow aria encore`

(B) `rise bow act encore`

(C) `rise bow act`

(D) `bow act aria`

(E) `bow aria encore`

Use the program description below for Questions 17–19.

A car dealer needs a program that will maintain an inventory of cars on his lot. There are four types of cars: sedans, station wagons, electric cars, and SUVs. The model, year, color, and price need to be recorded for each car, plus any additional features for the different types of cars. The program must allow the dealer to

- Add a new car to the lot.
- Remove a car from the lot.
- Correct any data that have been entered.
- Display information for any car.

17. The programmer decides to have these classes: `Car`, `Inventory`, `Sedan`, `SUV`, `ElectricCar`, and `StationWagon`. Which statement is true about the relationships between these classes and their attributes?

    I. There are no inheritance relationships between these classes.
    II. The `Inventory` class *has-a* list of `Car` objects.
    III. The `Sedan`, `SUV`, `ElectricCar`, and `StationWagon` classes are independent of each other.

    (A) I only
    (B) II only
    (C) III only
    (D) I and II only
    (E) II and III only

18. Suppose that the programmer decides to have a `Car` class and an `Inventory` class. The `Inventory` class will maintain a list of all the cars on the lot. Here are some of the methods in the program.

```
addCar          //adds a car to the lot
removeCar       //removes a car from the lot
displayCar      //displays all the features of a given car
setColor        //sets the color of a car to a given color
                //   (may be used to correct data)
getPrice        //returns the price of a car
displayAllCars  //displays features for every car on the lot
```

In each of the following, a class and a method are given. Which is the least suitable choice of class to be responsible for the given method?

(A) `Car, setColor`
(B) `Car, removeCar`
(C) `Car, getPrice`
(D) `Car, displayCar`
(E) `Inventory, displayAllCars`

19. Suppose `Car` is a superclass and `Sedan`, `StationWagon`, `ElectricCar`, and `SUV` are subclasses of `Car`. Which of the following is the most likely method of the `Car` class to be overridden by at least one of the subclasses (`Sedan`, `StationWagon`, `ElectricCar`, or `SUV`)?

(A) `setColor(newColor)   //sets color of Car to newColor`
(B) `getModel()           //returns model of Car`
(C) `displayCar()         //displays all features of Car`
(D) `setPrice(newPrice)   //sets price of Car to newPrice`
(E) `getYear()            //returns year of Car`

20. Consider the following segment of code.

```
String word = "conflagration";
int x = word.indexOf("flag");
String s = word.substring(0, x);
```

What will be the result of executing the above segment?

(A) A syntax error will occur.

(B) String `s` will be the empty string.

(C) String `s` will contain `"flag"`.

(D) String `s` will contain `"conf"`.

(E) String `s` will contain `"con"`.

21. A two-dimensional matrix `mat` with at least one row is initialized and will be traversed using a row-major (row-by-row, left-to-right) traversal. Which represents the last element accessed?

(A) `mat[mat.length][mat[0].length]`

(B) `mat[mat[0].length][mat.length]`

(C) `mat[mat.length - 1][mat[0].length - 1]`

(D) `mat[mat[0].length - 1][mat.length - 1]`

(E) `mat[mat.length - 1][mat.length - 1]`

22. A class of 30 students rated their computer science teacher on a scale of 1 to 10 (1 means awful and 10 means outstanding). The `responses` array is a 30-element integer array of the student responses. An 11-element array `freq` will count the number of occurrences of each response. For example, `freq[6]` will count the number of students who responded 6. The quantity `freq[0]` will not be used.

Here is a program that counts the students' responses and outputs the results.

```
public class StudentEvaluations
{
    public static void main(String args[])
    {
        int[] responses = {6,6,7,8,10,1,5,4,6,7,
                           5,4,3,4,4,9,8,6,7,10,
                           6,7,8,8,9,6,7,8,9,2};
        int[] freq = new int[11];
        for (int i = 0; i < responses.length; i++)
            freq[responses[i]]++;
        //output results
        System.out.print("rating" + "  " + "frequency\n");
        for (int rating = 1; rating < freq.length; rating++)
            System.out.print(rating + "  " +
                freq[rating] + "\n");
    }
}
```

Suppose the last entry in the initializer list for the `responses` array was incorrectly typed as 12 instead of 2. What would be the result of running the program?

(A) A rating of 12 would be listed with a frequency of 1 in the output table.

(B) A rating of 1 would be listed with a frequency of 12 in the output table.

(C) An `ArrayIndexOutOfBoundsException` would be thrown.

(D) A `StringIndexOutOfBoundsException` would be thrown.

(E) A `NullPointerException` would be thrown.

Questions 23–25 are based on the three classes below.

```
public class Employee
{
    private String name;
    private int employeeNum;
    private double salary, taxWithheld;

    public Employee(String aName, int empNum, double aSalary,
        double aTax)
    { /* implementation not shown */ }

    /** @return pre-tax salary */
    public double getSalary()
    { return salary; }

    public String getName()
    { return name; }

    public int getEmployeeNum()
    { return employeeNum; }

    public double getTax()
    { return taxWithheld; }

    public double computePay()
    { return salary - taxWithheld; }
}

public class PartTimeEmployee extends Employee
{
    private double payFraction;

    public PartTimeEmployee(String aName, int empNum, double aSalary,
        double aTax, double aPayFraction)
    { /* implementation not shown */ }

    public double getPayFraction()
    { return payFraction; }

    public double computePay()
    { return getSalary() * payFraction - getTax();}
}

public class Consultant extends Employee
```

23. The `computePay` method in the `Consultant` class redefines the `computePay` method of the `Employee` class to add a bonus to the salary after subtracting the tax withheld. Which represents correct `/*` ***implementation code*** `*/` of `computePay` for `Consultant`?

```
    private static final double BONUS = 5000;

    public Consultant(String aName, int empNum, double aSalary,
        double aTax)
    { /* implementation not shown */ }

    public double computePay()
    { /* implementation code */ }
}
```

I. `return super.computePay() + BONUS;`

II.
```
super.computePay();
return getSalary() + BONUS;
```

III. `return getSalary() - getTax() + BONUS;`

(A) I only
(B) II only
(C) III only
(D) I and III only
(E) I and II only

24. Consider these valid declarations in a client program.

```
Employee e = new Employee("Noreen Rizvi", 304, 65000, 10000);
Employee p = new PartTimeEmployee("Rafael Frongillo", 287, 40000,
    7000, 0.8);
Employee c = new Consultant("Dan Lepage", 694, 55000, 8500);
```

Which of the following method calls will cause an error?

(A) `double x = e.computePay();`
(B) `double y = p.computePay();`
(C) `String n = c.getName();`
(D) `int num = p.getEmployeeNum();`
(E) `double g = p.getPayFraction();`

25. Consider the `writePayInfo` method.

```
/** Writes Employee name and pay on one line. */
public static void writePayInfo(Employee e)
{ System.out.println(e.getName() + " " + e.computePay()); }
```

The following piece of code invokes this method.

```
Employee[] empList = new Employee[3];
empList[0] = new Employee("Lila Fontes", 1, 10000, 850);
empList[1] = new Consultant("Momo Liu", 2, 50000, 8000);
empList[2] = new PartTimeEmployee("Moses Wilks", 3, 25000, 3750,
    0.6);
for (Employee e : empList)
    writePayInfo(e);
```

What will happen when this code is executed?

(A) A `NullPointerException` will be thrown.

(B) An `ArrayIndexOutOfBoundsException` will be thrown.

(C) A compile-time error will occur, with the message that the `getName` method is not in the `Consultant` class.

(D) A compile-time error will occur, with the message that an instance of an `Employee` object cannot be created.

(E) A list of employees' names and corresponding pay will be written to the screen.

26. Consider an array `arr` that is initialized with `int` values. The following code segment stores in `count` the number of positive values in `arr`.

```
int count = 0, index = 0;
while (index < arr.length)
{
    if (arr[index] > 0)
        count++;
    index++;
}
```

Which of the following is equivalent to the above segment?

I.
```
int count = 0;
for (int num : arr)
{
    if (arr[num] > 0)
        count++;
}
```

II.
```
int count = 0;
for (int num : arr)
{
    if (num > 0)
        count++;
}
```

III.
```
int count = 0;
for (int i = 0; i < arr.length; i++)
{
    if (arr[i] > 0)
        count++;
}
```

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I and III only

27. A square matrix is declared as

```
int[][] mat = new int[SIZE][SIZE];
```

where `SIZE` is an appropriate integer constant. Consider the following method.

```
public static void mystery(int[][] mat, int value, int top, int left,
    int bottom, int right)
{
    for (int i = left; i <= right; i++)
    {
        mat[top][i] = value;
        mat[bottom][i] = value;
    }
    for (int i = top + 1; i <= bottom - 1; i++)
    {
        mat[i][left] = value;
        mat[i][right] = value;
    }
}
```

Assuming that there are no out-of-range errors, which best describes what method `mystery` does?

(A) Places `value` in corners of the rectangle with corners (`top, left`) and (`bottom, right`)

(B) Places `value` in the diagonals of the square with corners (`top, left`) and (`bottom, right`)

(C) Places `value` in each element of the rectangle with corners (`top, left`) and (`bottom, right`)

(D) Places `value` in each element of the border of the rectangle with corners (`top, left`) and (`bottom, right`)

(E) Places `value` in the topmost and bottommost rows of the rectangle with corners (`top, left`) and (`bottom, right`)

28. Consider the following declaration.

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

Which of the following code segments will place the integers 1 to 10, in any order, in the empty list?

I.
```
for (int i = 0; i < 10; i++)
    list.add(i + 1);
```

II.
```
for (int i = 0; i < 10; i++)
    list.add(i, i + 1);
```

III.
```
for (int i = 9; i >= 0; i--)
    list.add(i, i + 1);
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

29. Assume that a `Book` class has a `compareTo` method in which, if `b1` and `b2` are `Book` objects, `b1.compareTo(b2)` is a negative integer if `b1` is less than `b2`, a positive integer if `b1` is greater than `b2`, and `0` if `b1` equals `b2`. The following method is intended to return the index of the "smallest" book—namely, the book that would appear first in a sorted list of `Book` objects.

```
/** Precondition:
 *    - books is initialized with Book objects.
 *    - books.length > 0.
 */
public static int findMin(Book[] books)
{
    int minPos = 0;
    for (int index = 1; index < books.length; index++)
    {
        if ( /* condition */ )
        {
            minPos = index;
        }
    }
    return minPos;
}
```

Which of the following should be used to replace `/* condition */` so that `findMin` works as intended?

(A) `books[index] < books[minPos]`

(B) `books[index] > books[minPos]`

(C) `books[index].compareTo(books[minPos]) > 0`

(D) `books[index].compareTo(books[minPos]) >= 0`

(E) `books[index].compareTo(books[minPos]) < 0`

30. Refer to the static method `removeNegs` shown below.

```
/** Precondition:  list is an ArrayList<Integer>.
 *  Postcondition: All negative values have been removed from list.
 */
public static void removeNegs(ArrayList<Integer> list)
{
    int index = 0;
    while (index < list.size())
    {
        if (list.get(index).intValue() < 0)
        {
            list.remove(index);
        }
        index++;
    }
}
```

For which of the following lists will the method not work as intended?

(A) 6 −1 −2 5

(B) −1 2 −3 4

(C) 2 4 6 8

(D) −3

(E) 1 2 3 −8

31. A sorted list of 120 integers is to be searched to determine whether the value 100 is in the list. Assuming that the most efficient searching algorithm is used, what is the maximum number of elements that must be examined?

(A) 7
(B) 8
(C) 20
(D) 100
(E) 120

32. Consider a sorted array `arr` of *n* elements, where *n* is large and *n* is even. Under which conditions will a sequential search of `arr` be faster than a binary search?

I. The target is not in the list.

II. The target is in the first position of the list.

III. The target is in `arr[1 + n/2]`.

(A) I only
(B) II only
(C) III only
(D) I and III only
(E) II and III only

33. Refer to the following data field and method.

```
private int[] arr;

/** Precondition: arr.length > 0 and index < arr.length. */
public void remove(int index)
{
    int[] b = new int[arr.length - 1];
    int count = 0;
    for (int i = 0; i < arr.length; i++)
    {
        if (i != index)
        {
            b[count] = arr[i];
            count++;
        }
    }
    /* assertion */
    arr = b;
}
```

Which of the following assertions is true when the `/*` ***assertion*** `*/` line is reached during execution of `remove`?

(A) `b[k] == arr[k]` for `0 <= k < arr.length`.

(B) `b[k] == arr[k + 1]` for `0 <= k < arr.length`.

(C) `b[k] == arr[k]` for `0 <= k <= index`, and
`b[k] == arr[k + 1]` for `index < k < arr.length - 1`.

(D) `b[k] == arr[k]` for `0 <= k < index`, and
`b[k] == arr[k + 1]` for `index <= k < arr.length - 1`.

(E) `b[k] == arr[k]` for `0 <= k < index`, and
`b[k] == arr[k + 1]` for `index <= k < arr.length`.

34. Consider the following code segment.

```
for (int n = 25; n >= 0; n /= 2)
    System.out.println(n);
```

When the segment is executed, how many passes through the `for` loop will there be?

(A) Fewer than 5

(B) Between 5 and 12, inclusive

(C) Between 13 and 25, inclusive

(D) Between 26 and 100, inclusive

(E) More than 100

Questions 35–37 refer to the `TennisPlayer`, `GoodPlayer`, and `WeakPlayer` classes below. These classes are to be used in a program to simulate a game of tennis.

```
public class TennisPlayer
{
    private String name;

    public TennisPlayer(String aName)
    { name = aName; }

    public boolean serve()
    { /* implementation not shown */ }
}

public class GoodPlayer extends TennisPlayer
{
    public GoodPlayer(String aName)
    { /* implementation not shown */ }

    public boolean serve()
    { /* implementation not shown */ }
}

public class WeakPlayer extends TennisPlayer
{
    public WeakPlayer(String aName)
    { /* implementation not shown */ }

    /** Returns true if serve is in (45% probability),
     *  false if serve is out (55% probability).
     */
    public boolean serve()
    { /* implementation not shown */ }
}
```

35. Which of the following declarations will cause an error? You may assume all the constructors are correctly implemented.

(A) `WeakPlayer t = new TennisPlayer("Smith");`

(B) `TennisPlayer g = new GoodPlayer("Jones");`

(C) `TennisPlayer w = new WeakPlayer("Henry");`

(D) `TennisPlayer p = null;`

(E) `WeakPlayer q = new WeakPlayer("Grady");`

36. Refer to the `serve` method in the `WeakPlayer` class.

```
/** Returns true if serve is in (45% probability),
 *  false if serve is out (55% probability).
 */
public boolean serve()
{ /* implementation */ }
```

Which of the following replacements for /∗ ***implementation*** ∗/ satisfy the postcondition of the `serve` method?

I.
```
double value = Math.random();
return value >= 0 || value < 0.45;
```

II.
```
double value = Math.random();
return value < 0.45;
```

III.
```
int val = (int) (Math.random() * 100);
return val < 45;
```

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I, II, and III

37. Consider the following class definition.

```
public class Beginner extends WeakPlayer
{
    private double costOfLessons;

    //methods of Beginner class
        ...
}
```

Refer to the following declarations and method in a client program.

```
TennisPlayer w = new WeakPlayer("Harry");
TennisPlayer b = new Beginner("Dick");
Beginner bp = new Beginner("Ted");

public void giveEncouragement(WeakPlayer t)
{ /* implementation not shown */ }
```

Which of the following method calls will cause an error?

I. `giveEncouragement(w);`

II. `giveEncouragement(b);`

III. `giveEncouragement(bp);`

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

38. A matrix class that manipulates matrices contains the following declaration.

```
private int[][] mat = new int[numRows][numCols];
```

Consider the following method that alters matrix `mat`.

```
public void doSomething()
{
    int width = mat[0].length;
    int numRows = mat.length;
    for (int row = 0; row < numRows; row++)
        for (int col = 0; col < width/2; col++)
            mat[row][col] = mat[row][width - 1 - col];
}
```

If `mat` has current value

```
1 2 3 4 5 6
1 3 5 7 9 11
```

what will the value of `mat` be after a call to `doSomething`?

(A)
```
1 2 3 3 2 1
1 3 5 5 3 1
```

(B)
```
 6 5 4 4 5 6
11 9 7 7 9 11
```

(C)
```
 6 5 4 3 2 1
11 9 7 5 3 1
```

(D)
```
1 2 3 4 5 6
1 2 3 4 5 6
```

(E)
```
1 3 5 7 9 11
1 3 5 7 9 11
```

Questions 39 and 40 refer to the following information.

Consider an array `arr` that is sorted in increasing order, and method `findMost` given below. Method `findMost` is intended to find the value in the array that occurs most often. If every value occurs exactly once, `findMost` should return `-1`. If there is more than one value that occurs the most, `findMost` should return any one of those. For example, if `arr` contains the values `[1,5,7,7,10], findMost` should return `7`. If `arr` contains `[2,2,2,7,8,8,9,9,9]`, `findMost` should return `2` or `9`. If `arr` contains `[1,2,7,8]`, `findMost` should return `−1`.

```
/** Precondition:  arr sorted in increasing order.
 */
public static int findMost(int[] arr)
{
    int index = 0;
    int count = 1;
    int maxCountSoFar = 1;
    int mostSoFar = arr[0];
    while (index < arr.length - 1)
    {
        while (index < arr.length - 1 &&
                arr[index] == arr[index + 1])
        {
            count++;
            index++;
        }
        if (count > maxCountSoFar)
        {
            maxCountSoFar = count;
            mostSoFar = arr[index];
        }
        index++;
    }
    if (maxCountSoFar == 1)
        return -1;
    else
        return mostSoFar;
}
```

39. The method `findMost` does not always work as intended. An *incorrect* result will be returned if `arr` contains the values

(A) `[1,2,3,4,5]`

(B) `[6,6,6,6]`

(C) `[1,2,2,3,4,5]`

(D) `[1,1,3,4,5,5,5,7]`

(E) `[2,2,2,4,5,5]`

40. Which of the following changes should be made so that method `findMost` will work as intended?

(A) Insert the statement `count = 1;` between Lines 20 and 21.

(B) Insert the statement `count = 1;` between Lines 21 and 22.

(C) Insert the statement `count = 1;` between Lines 16 and 17.

(D) Insert the statement `count = 0;` between Lines 23 and 24.

(E) Insert the statement `count = 1;` between Lines 23 and 24.

END OF SECTION I

COMPUTER SCIENCE A
SECTION II

Time—1 hour and 30 minutes
4 Questions

DIRECTIONS: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Write your answers in the separate Free-Response booklet provided.

NOTES:

- Assume that the classes in the Quick Reference have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods will not receive full credit.

1. A `WordSet`, whose partial implementation is shown in the class declaration below, stores a set of `String` objects in no particular order and contains no duplicates. Each word is a sequence of capital letters only.

```
public class WordSet
{
    /** Constructor initializes set to empty. */
    public WordSet()
    { /* implementation not shown */ }

    /** Returns the number of words in set. */
    public int size()
    { /* implementation not shown */ }

    /** Adds word to set.
     */
    public void insert(String word)
    { /* implementation not shown */ }

    /** Removes word from set if present, else does nothing.
     */
    public void remove(String word)
    { /* implementation not shown */ }

    /** Returns kth word in alphabetical order, where 1 <= k <= size().
     */
    public String findkth(int k)
    { /* implementation not shown */ }

    /** Returns true if set contains word, false otherwise. */
    public boolean contains(String word)
    { /* implementation not shown */ }

    //Other instance variables, constructors, and methods are not shown.
}
```

The `findkth` method returns the *k*th `word` in alphabetical order in the set, even though the implementation of `WordSet` may not be sorted. The number *k* ranges from 1 (corresponding to first in alphabetical order) to *N*, where *N* is the number of words in the set. For example, if `WordSet s` stores the words `{"GRAPE", "PEAR", "FIG", "APPLE"}`, here are the values when `s.findkth(k)` is called.

k	values of `s.findkth(k)`
1	APPLE
2	FIG
3	GRAPE
4	PEAR

Class information for this question

```
public class WordSet

public WordSet()
public int size()
public void insert(String word)
public void remove(String word)
public String findkth(int k)
public boolean contains(String word)
```

(a) Write a client method `countA` that returns the number of words in `WordSet s` that begin with the letter “A.” In writing `countA`, you may call any of the methods of the `WordSet` class. Assume that the methods work as specified.

Complete method `countA`.

```
/**  Returns the number of words in s that begin with "A".
 */
public static int countA(WordSet s)
```

(b) Write a client method `removeA` that removes all words that begin with “A” from a non-null `WordSet`. If there are no such words in `s`, then `removeA` does nothing. In writing `removeA`, you may call method `countA` specified in part (a). Assume that `countA` works as specified, regardless of what you wrote in part (a).

Complete method `removeA`.

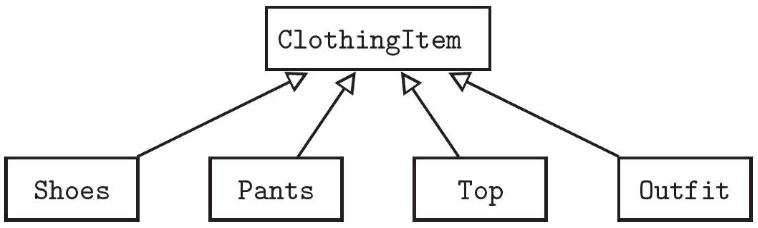
```
/** Removes from WordSet s all words that begin with the letter "A".
 *  Precondition:  WordSet is not null.
 *  Postcondition: WordSet s contains no words that begin with
 *                 "A", but is otherwise unchanged.
 */
public static void removeA(WordSet s)
```

2. A clothing store sells shoes, pants, and tops. The store also allows a customer to buy an “outfit,” which consists of three items: one pair of shoes, one pair of pants, and one top.

   Each clothing item has a description and a price. The four types of clothing items are represented by the four classes `Shoes`, `Pants`, `Top`, and `Outfit`. All four classes are subclasses of a `ClothingItem` class, shown below.

```
public class ClothingItem
{
    private String description;
    private double price;

    public ClothingItem()
    {
        description = "";
        price = 0;
    }

    public ClothingItem(String descr, double aPrice)
    {
        description = descr;
        price = aPrice;
    }

    public String getDescription()
    { return description; }

    public double getPrice()
    { return price; }
}
```

The following diagram shows the relationship between the `ClothingItem` class and the `Shoes`, `Pants`, `Top`, and `Outfit` classes.



The store allows customers to create `Outfit` clothing items, each of which includes a pair of shoes, pants, and a top. The description of the outfit consists of the description of the shoes, pants, and top, in that order, separated by "/" and followed by a space and "outfit". The price of an outfit is calculated as follows. If the sum of the prices of any two items equals or exceeds $\$100$, there is a 25% discount on the sum of the prices of all three items. Otherwise there is a 10% discount.

For example, an outfit consisting of sneakers ($\$40$), blue jeans ($\$50$), and a T-shirt ($\$10$) would have the name "sneakers/blue jeans/T-shirt outfit" and a price of $0.90(40 + 50 + 10) = \$90.00$. An outfit consisting of loafers ($\$50$), cutoffs ($\$20$), and dress-shirt ($\$60$) would have the description "loafers/cutoffs/dress-shirt outfit" and price $0.75(50 + 20 + 60) = \$97.50$.

Write the `Outfit` subclass of `ClothingItem`. Your implementation must have just one constructor that takes three parameters representing a pair of shoes, pants, and a top, in that order.

A client class that uses the `Outfit` class should be able to create an outfit, get its description, and get its price. Your implementation should be such that the client code has the following behavior:

```
Shoes shoes;
Pants pants;
Top top;
/* Code to initialize shoes, pants, and top */

ClothingItem outfit =
    new Outfit (shoes, pants, top); //Compiles without error
ClothingItem outfit =
    new Outfit (pants, shoes, top); //Compile-time error
ClothingItem outfit =
    new Outfit (shoes, top, pants); //Compile-time error
```

3. Consider a note keeper object that is designed to store and manipulate a list of short notes. Here are some typical notes:

```
pick up drycleaning
special dog chow
car registration
dentist Monday
dog license
```

A note is represented by the following class.

```
public class Note
{
    /** Returns a one-line note. */
    public String getNote()
    { /* implementation not shown */ }

    //There may be instance variables, constructors, and methods
    //that are not shown.
}
```

A note keeper is represented by the `NoteKeeper` class shown below.

```
public class NoteKeeper
{
    /** The list of notes */
    private ArrayList<Note> noteList;

    /** Prints all notes in noteList, as described in part(a).
     */
    public void printNotes()
    { /* to be implemented in part (a) */}

    /** Removes all notes with specified string from noteList,
     *  as described in part (b).
     *  If none of the notes in noteList contains the given string,
     *  the list remains unchanged.
     */
    public void removeNotes(String str)
    { /* to be implemented in part (b) */ }

    //There may be instance variables, constructors, and methods
    //that are not shown.
}
```

(a) Write the `NoteKeeper` method `printNotes`. This method prints all of the notes in `noteList`, one per line, and numbers the notes, starting at 1. The output should look like this.

```
1. pick up drycleaning
2. special dog chow
3. car registration
4. dentist Monday
5. dog license
```

Complete method `printNotes`.

```
/** Prints all notes in noteList, as described in part(a).
 */
public void printNotes()
```

(b) Write the `NoteKeeper` method `removeNotes`. Method `removeNotes` removes all notes from `noteList` that contain the string specified by the parameter. The ordering of the remaining notes should be left unchanged. For example, suppose that a `NoteKeeper` variable, `notes`, has a `noteList` containing the following.

```
[pick up drycleaning, special dog chow, car registration,
 dentist Monday, dog license]
```

The method call `notes.removeNotes("dog")` should modify the `noteList` of `notes` to be

```
[pick up drycleaning, car registration, dentist Monday]
```

The method call `notes.removeNotes("cow")` should leave the list shown above unchanged.

Here's another example. If `noteList` contains

```
[pick up car, buy carrots, dog license, carpet cleaning]
```

the method call `notes.removeNotes("car")` should modify the `noteList` of notes to be

```
[dog license]
```

Complete method `removeNotes`.

```
/** Removes all notes with specified string from noteList,
 *  as described in part (b).
 *  If none of the notes in noteList contains the given string,
 *  the list remains unchanged.
 */
public void removeNotes(String str)
```

4. Consider the problem of keeping track of the available seats in a theater. Theater seats can be represented with a two-dimensional array of integers, where a value of `0` shows a seat is available, while a value of `1` indicates that the seat is occupied. For example, the array below shows the current seat availability for a show in a small theater.

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	0	1	1	0	1
[1]	0	1	0	1	0	1
[2]	1	0	0	0	0	0

The seat at slot `[1][3]` is taken, but seat `[0][4]` is still available.

A show can be represented by the `Show` class shown below.

```
public class Show
{
    /** The seats for this show */
    private int[][] seats;

    private final int SEATS_PER_ROW = <some integer value>;
    private final int NUM_ROWS = <some integer value>;

    /** Reserve two adjacent seats and return true if this was
     *  successfully done, false otherwise, as described in part (a).
     */
    public boolean twoTogether()
    { /* to be implemented in part (a) */ }

    /** Return the lowest seat number in the specified row for a
     *  block of seatsNeeded empty adjacent seats, as described in part (b).
     */
    public int findAdjacent(int row, int seatsNeeded)
    { /* to be implemented in part (b) */ }

    //There may be instance variables, constructors, and methods
    //that are not shown.
}
```

(a) Write the `Show` method `twoTogether`, which reserves two adjacent seats and returns `true` if this was successfully done. If it is not possible to find two adjacent seats that are unoccupied, the method should leave the show unchanged and return `false`. For example, suppose this is the state of a show.

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	0	1	1	0	1
[1]	0	1	0	1	0	1
[2]	1	0	0	0	1	1

A call to `twoTogether` should return `true`, and the final state of the show could be any one of the following three configurations.

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	1	1	1	1	0	1
[1]	0	1	0	1	0	1
[2]	1	0	0	0	1	1

OR

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	0	1	1	0	1
[1]	0	1	0	1	0	1
[2]	1	1	1	0	1	1

OR

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	0	1	1	0	1
[1]	0	1	0	1	0	1
[2]	1	0	1	1	1	1

For the following state of a show, a call to `twoTogether` should return `false` and leave the two-dimensional array as shown.

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	1	0	1	1	0
[1]	1	1	0	1	0	1
[2]	0	1	1	1	1	1

Class information for this question

<u>`public class Show`</u>

```
private int[][] seats
private final int SEATS_PER_ROW
private final int NUM_ROWS
public boolean twoTogether()
public int findAdjacent(int row, int seatsNeeded)
```

Complete method `twoTogether`.

```
/** Reserve two adjacent seats and return true if this was
 *  successfully done, false otherwise, as described in part (a).
 */
public boolean twoTogether()
```

(b) Write the `Show` method `findAdjacent`, which finds the lowest seat number in the specified row for a specified number of empty adjacent seats. If no such block of empty seats exists, the `findAdjacent` method should return `-1`. No changes should be made to the state of the show, irrespective of the value returned.

For example, suppose the diagram of seats is as shown.

$$\begin{array}{ccccccc}
 & [0] & [1] & [2] & [3] & [4] & [5] \\
[0] & 0 & 1 & 1 & 0 & 0 & 0 \\
[1] & 0 & 0 & 0 & 0 & 1 & 1 \\
[2] & 1 & 0 & 0 & 1 & 0 & 0
\end{array}$$

The following table shows some examples of calling `findAdjacent` for `show`.

Method call	Return value
`show.findAdjacent(0,3)`	3
`show.findAdjacent(1,3)`	`0` or 1
`show.findAdjacent(2,2)`	1 or 4
`show.findAdjacent(1,5)`	-1

Complete method `findAdjacent`.

```
/** Return the lowest seat number in the specified row for a
 *  block of seatsNeeded empty adjacent seats, as described in part (b).
 */
public int findAdjacent(int row, int seatsNeeded)
```

STOP

END OF EXAM

ANSWER KEY

Practice Test 2

Section I

1. E
2. B
3. C
4. D
5. A
6. A
7. D
8. E
9. A
10. D
11. C
12. E
13. C
14. B
15. E
16. A
17. E

18.	**B**
19.	**C**
20.	**E**
21.	**C**
22.	**C**
23.	**D**
24.	**E**
25.	**E**
26.	**D**
27.	**D**
28.	**D**
29.	**E**
30.	**A**
31.	**A**
32.	**B**
33.	**D**
34.	**E**
35.	**A**
36.	**D**
37.	**D**
38.	**B**
39.	**E**
40.	**B**

Answer Explanations

Section I

1. **(E)** The string parameter in the line of code uses two escape characters:
   `\"`, which means print a double quote.
   `\n`, which means print a newline character (i.e., go to the next line).

2. **(B)** The intent of the programmer is to have overloaded `getValue` methods in `SomeClass`. Overloaded methods have different signatures, where the signature of a method includes the name and parameter types only. Thus, the signature of the original method is `getValue(int)`. The signature in header I is `getValue()`. The signature in header II is `getValue(int)`. The signature in header III is `getValue(double)`. Since the signature in header II is the same as that of the given method, the compiler will flag it and say that the method already exists in `SomeClass`. Note: The return type of a method is not included in its signature.

3. **(C)** The expression `(int)(Math.random() * 49)` produces a random integer from 0 through 48. (Note that 49 is the number of possibilities for `num`.) To shift this range from 2 to 50, add 2 to the expression.

4. **(D)** Short-circuit evaluation of the boolean expression will occur. The expression `(num != 0)` will evaluate to `false`, which makes the entire boolean expression `false`. Therefore the expression `(score/num > SOME_CONSTANT)` will not be evaluated. Hence no division by zero will occur, and there will be no `ArithmeticException` thrown. When the boolean expression has a value of `false`, only the `else` part of the statement, ***statement2***, will be executed.

5. **(A)** The values of `k` are, consecutively, `4, 3, 2`, and `1`. The values of `randPos` are, consecutively, `3, 2, 0`, and `0`. Thus, the sequence of swaps and corresponding states of `nums` will be:

```
swap nums[4] and nums[3]        8 7 6 4 5
swap nums[3] and nums[2]        8 7 4 6 5
swap nums[2] and nums[0]        4 7 8 6 5
swap nums[1] and nums[0]        7 4 8 6 5
```

Thus, the element in `nums[2]` is `8`.

6. **(A)** A matrix is stored as an array of arrays; that is, each row is an array. Therefore it is correct to call a method with an array parameter for each row, as is done in segment I. Segment II fails because `mat` is not an array of columns. The segment would cause an error, since `mat[col]` refers to a *row*, not a column. (If the number of rows were less than the number of columns, the method would throw an `ArrayIndexOutOfBoundsException`. If the number of rows were greater than the number of columns, the method would correctly assign the value `100` to the first `n` rows, where `n` is the number of columns. The rest of the rows would retain the values before execution of the method.) Segment III fails because this is incorrect usage of an enhanced `for` loop, which should not be used to assign new elements in the matrix. The matrix remains unchanged.
7. **(D)** Declaration I fails because it fails this test: `Cereal` *is-a* `WheatCereal`? No. Notice that declarations II and III pass this test: `Cereal` *is-a* `Cereal`? Yes. `RiceCereal` *is-a* `Cereal`? Yes.
8. **(E)** All satisfy the *is-a* test! `Class2` *is-a* `Class1`. `Class3` *is-a* `Class2`. `Class3` *is-a* `Class1`. Note: Since `Class3` is a subclass of `Class2`, it automatically implements any interfaces implemented by `Class2`, its superclass.
9. **(A)** Method call I works because `Class3` inherits all the methods of `Class1` and `Class2`. Method call II fails because `Class2` does not inherit the methods of `Class3`, its subclass. Method call III uses a parameter that fails the *is-a* test: `ob2` is *not* a `Class3`, which the parameter requires.

10. **(D)** After each execution of the loop body, `n` is divided by 2. Thus, the loop will produce output when `n` is 50, 25, 12, 6, 3, and 1. The final value of `n` will be 1 / 2, which is 0, and the test will fail.

11. **(C)** Statement III will cause an `IndexOutOfBoundsException` because there is no slot 4. The final element, `"Luis"`, is in slot 3. Statement I is correct: It replaces the string `"Harry"` with the string `"6"`. It may look peculiar in the list, but the syntax is correct. Statement II looks like it may be out of range because there is no slot 4. It is correct, however, because you must be allowed to add an element to the end of the list.

12. **(E)** The effect of the given algorithm is to raise `n` to the 8th power.

    When $i = 1$, the result is `n * n` = n^2.

    When $i = 2$, the result is $n^2 * n^2 = n^4$.

    When $i = 3$, the result is $n^4 * n^4 = n^8$.

13. **(C)** The method traverses `nums`, starting at position `0`, and returns the current position the first time it finds an odd value. This implies that all values in positions `0` through the current index − 1 contained even numbers.

14. **(B)** Since `n == 6` fails the two base case tests, the method call `mystery(6)` returns `6 + mystery(5)`. Since 5 satisfies the second base case test, `mystery(5)` returns 5, and there are no more recursive calls. Thus, `mystery(6)` = 6 + 5 = 11.

15. **(E)** In order for /* ***body of loop*** */ not to be executed, the test must be false the first time it is evaluated. A compound OR test will be false if and only if both pieces of the test are false. Thus, choices B and C are insufficient. Choice D fails because it guarantees that both pieces of the test will be *true*. Choice A is wrong because /* ***body of loop*** */ may be executed many times, until the computer runs out of memory (an infinite loop!).

16. **(A)** When `p.act()` is called, the `act` method of `Singer` is executed. This is an example of polymorphism. The first line prints `rise`. Then `super.act()` goes to the `act` method of `Performer`, the superclass. This prints `bow` and then calls `perform()`. Again, using polymorphism, the `perform` method in `Singer` is called, which prints `aria`. Now, completing the `act` method of `Singer, encore` is printed. The result?

```
rise bow aria encore
```

17. **(E)** Statement I is false: The `Sedan, StationWagon`, and `SUV` classes should all be subclasses of `Car`. Each one satisfies the *is-a* `Car` relationship. Statement II is true: The main task of the `Inventory` class should be to keep an updated list of `Car` objects. Statement III is true: A class is independent of another class if it does not require that class to implement its methods.

18. **(B)** The `Inventory` class is responsible for maintaining the list of all cars on the lot. Therefore methods like `addCar, removeCar`, and `displayAllCars` must be the responsibility of this class. The `Car` class should contain the `setColor, getPrice`, and `displayCar` methods, since all these pertain to the attributes of a given `Car`.

19. **(C)** Each subclass may contain additional attributes for the particular type of car that are not in the `Car` superclass. Since `displayCar` displays all features of a given car, this method should be overridden to display the original plus additional features.

20. **(E)** The expression `word.indexOf("flag")` returns the index of the first occurrence of `"flag"` in the calling string, `word`. Thus, `x` has value 3. (Recall that the first character in `word` is at index 0.) The method call `word.substring(0, x)` is equivalent to `word.substring(0, 3)`, which returns the substring in `word` from 0 to 2—namely, `"con"`. The character at index 3 is not included.

21. **(C)** The number of rows in `mat` is given by `mat.length`. The indexes of the rows range from `0` to `mat.length - 1`. The

number of columns in `mat` is given by `mat[0].length`. You can think of this as the length of the array represented by `mat[0]`, the first row. (Note that all the rows have the same length, but it is wise to use `mat[0]`, since you know that the matrix has at least one row. For example, `mat[1]` may be out of bounds.) The indexes of the columns range from `0` to `mat[0].length - 1`. The last element to be accessed in a row-by-row (or column-by-column) traversal is in the bottom right corner—namely, `mat[mat.length - 1][mat[0].length - 1]`.

22. **(C)** If the `responses` array contained an invalid value like `12`, the program would attempt to add 1 to `freq[12]`. This is out of bounds for the `freq` array.

23. **(D)** Implementation I calls `super.computePay()`, which is equivalent to the `computePay` method in the `Employee` superclass. The method returns the quantity `(salary - taxWithheld)`. The `BONUS` is then correctly added to this expression, as required. Implementation III correctly uses the public accessor methods `getSalary` and `getTax` that the `Consultant` class has inherited. Note that the `Consultant` class does not have direct access to the private instance variables `salary` and `taxWithheld`. Implementation II incorrectly returns the salary plus `BONUS`—there is no tax withheld. The expression `super.computePay()` returns a value equal to salary minus tax. But this is neither stored nor included in the `return` statement.

24. **(E)** Note that `p` is declared to be of type `Employee`, and the `Employee` class does not have a `getPayFraction` method. To avoid the error, `p` must be cast to `PartTimeEmployee` as follows:

```
double g = ((PartTimeEmployee) p).getPayFraction();
```

25. **(E)** The code does exactly what it looks like it should. The `writePayInfo` parameter is of type `Employee` and each element of the `empList` array *is-a* `Employee` and therefore does not need to be cast to its actual instance type. This is an example of polymorphism, in which the appropriate `computePay` method is

chosen during run time. There is no `ArrayIndexOutOfBoundsException` (choice B) since the array is accessed using an enhanced `for` loop. None of the array elements is null; therefore, there is no `NullPointerException` (choice A). Choice C won't happen because the `getName` method is inherited by both the `Consultant` and `PartTimeEmployee` classes. Choice D would occur if the `Employee` superclass were abstract, but it's not.

26. **(D)** Segment I is incorrect because `num` is not an index in the loop: It is a value in the array. Thus, the correct test is `if (num > 0)`, which is correctly used in segment II. Segment III is a regular `for` loop, exactly equivalent to the given `while` loop.

27. **(D)** The first `for` loop places `value` in the top and bottom rows of the defined rectangle. The second `for` loop fills in the remaining border elements on the sides. Note that the `top + 1` and `bottom - 1` initializer and terminating conditions avoid filling in the corner elements twice.

28. **(D)** Segment I works because each `int` value, `i + 1`, is autoboxed in an `Integer`. Segment II works similarly using an overloaded version of the `add` method. The first parameter is the index and the second parameter is the element to be added. Segment III fails because an attempt to add an element to an empty list at a position other than `0` will cause an `IndexOutOfBoundsException` to be thrown.

29. **(E)** Eliminate choices A and B: When comparing `Book` objects, you cannot use simple inequality operators; you *must* use `compareTo`. For the calling object to be *less than* the parameter object, use the *less than* 0 test (a good way to remember this!).

30. **(A)** Method `removeNegs` will not work whenever there are consecutive negative values in the list. This is because removal of an element from an `ArrayList` causes the elements to the right of it to be shifted left to fill the "hole." The index in the given algorithm, however, always moves one slot to the

right. Therefore in choice A, when `-1` is removed, `-2` will be passed over, and the final list will be `6 -2 5`.

31. **(A)** If the list is sorted, a binary search is the most efficient algorithm to use. Binary search chops the current part of the array being examined in half, until you have found the element you are searching for, or there are no elements left to look at. In the worst case, you will need to divide by 2 seven times:

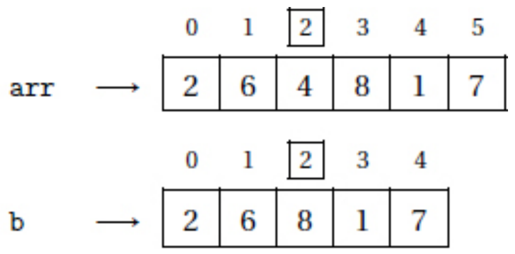
$$\begin{gathered} 120/2 \rightarrow 60 \\ 60/2 \rightarrow 30 \\ 30/2 \rightarrow 15 \\ 15/2 \rightarrow 7 \\ 7/2 \rightarrow 3 \\ 3/2 \rightarrow 1 \\ 1/2 \rightarrow 0 \end{gathered}$$

Shortcut: Round 120 to the nearest power of 2, 128. Since $128 = 2^7$, the number of comparisons in the worst case equals the exponent, 7.

32. **(B)** For a sequential search, all *n* elements will need to be examined. For a binary search, the array will be chopped in half a maximum of $\log_2 n$ times. When the target is in the first position of the list, a sequential search will find it in the first comparison. The binary search, which examines a middle element first, will not. Condition I is a worst case situation for the sequential search. It will require far more comparisons than a binary search. Condition III is approximately the middle of the list, but it won't be found on the first try of the binary search. (The first try examines `arr[n/2]`.) Still, the target *will* be located within fewer than log*n* tries, whereas the sequential search will need more than *n*/2 tries.

33. **(D)** The `remove` method removes from `arr` the element `arr[index]`. It does this by copying all elements from `arr[0]` up

to but not including `arr[index]` into array `b`. Thus, `b[k] == arr[k]` for `0 <= k < index` is true. Then it copies all elements from `arr[index + 1]` up to and including `arr[arr.length - 1]` into `b`. Since no gaps are left in `b`, `b[k] == arr[k + 1]` for `index <= k < arr.length - 1`. The best way to see this is with a small example. If `arr` is 2, 6, 4, 8, 1, 7, and the element at `index` 2 (namely, the 4) is to be removed, here is the picture:



```
b[0] == arr[0]
b[1] == arr[1]
b[2] == arr[3]
b[3] == arr[4]
b[4] == arr[5]
```

Notice that `arr.length` is 6, but `k` ends at 4.

34. **(E)** The segment is an example of an infinite loop. Here are successive values of `n`, which is updated with successive divisions by two using the `div` operator:

```
25, 12, 6, 3, 1, 0, 0, 0, …
```

Note that `0/2` equals `0`, which never fails the `n >= 0` test.

35. **(A)** Choice A is illegal because it fails this test: `TennisPlayer` *is-a* `WeakPlayer`? The answer is no, not necessarily.

36. **(D)** The statement

```
double value = Math.random();
```

generates a random `double` in the range $0 \le$ `value` < 1. Since random doubles are uniformly distributed in this interval, 45 percent of the time you can expect `value` to be in the range $0 \le$ `value` < 0.45. Therefore, a test for `value` in this range can be a test for whether the serve of a `WeakPlayer` went in. Since `Math.random()` never returns a negative number, the test in implementation II, `value < 0.45`, is sufficient. The test in implementation I would be correct if `||` were changed to `&&` ("or" changed to "and"—both parts must be true). Implementation III also works. The expression

```
(int) (Math.random() * 100)
```

returns a random integer from 0 to 99, each equally likely. Thus, 45 percent of the time, the integer `val` will be in the range $0 \le$ `val` ≤ 44. Therefore, a test for `val` in this range can be used to test whether the serve was in.

37. **(D)** Method calls I and II will each cause a compile-time error: The parameter must be of type `WeakPlayer`, but `w` and `b` are declared to be of type `TennisPlayer`. Each of these choices can be corrected by casting the parameter to `WeakPlayer`. Method call III works because `bp` *is-a* `WeakPlayer`.

38. **(B)** The method copies the elements from columns `3, 4`, and `5` into columns `2, 1`, and `0`, respectively, as if there were a vertical mirror down the middle of the matrix. To see this, here are the values for the given matrix: `width = 6, width/2 = 3, numRows = 2`. The variable `row` goes from `0` to `1` and `column` goes from `0` to `2`. The element assignments are

```
mat[0][0] = mat[0][5]
mat[0][1] = mat[0][4]
mat[0][2] = mat[0][3]
mat[1][0] = mat[1][5]
mat[1][1] = mat[1][4]
mat[1][2] = mat[1][3]
```

39. **(E)** In choice E, `findMost` returns the value `5`. This is because `count` has not been reset to `1`, so that when `5` is encountered, the test `count>maxCountSoFar` is `true`, causing `mostSoFar` to be

incorrectly reassigned to `5`. In choices A, B, and C, the outer `while` loop is not entered again, since a second run of equal values doesn't exist in the array. So `mostSoFar` comes out with the correct value. In choice D, when the outer loop is entered again, the test `count>maxCountSoFar` just happens to be `true` anyway and the correct value is returned. The algorithm fails whenever a new string of equal values is found whose length is shorter than a previous string of equal values.

40. **(B)** The `count` variable must be reset to `1` as soon as `index` is incremented in the outer `while` loop, so that when a new run of equal values is found, `count` starts out as `1`.

Section II

1. (a)

```
public static int countA(WordSet s)
{
    int count = 0;
    while (count < s.size() &&
            s.findkth(count + 1).substring(0, 1).equals("A"))
        count++;
    return count;
}
```

Alternatively,

```
public static int countA(WordSet s)
{
    boolean done = false;
    int count = 0;
    while (count < s.size() && !done)
    {
        String nextWord = s.findkth(count + 1);
        if (nextWord.substring(0,1).equals("A"))
            count++;
        else
            done = true;
    }
    return count;
}
```

(b)

```
public static void removeA(WordSet s)
{
    int numA = countA(s);
    for (int i = 1; i <= numA; i++)
        s.remove(s.findkth(1));
}
```

Alternatively,

```
public static void removeA(WordSet s)
{
    while (s.size() != 0 &&
            s.findkth(1).substring(0, 1).equals("A"))
        s.remove(s.findkth(1));
}
```

Note

- In part (a), to test whether a `word` starts with `"A"`, you must compare the first letter of `word`, that is, `word.substring(0,1)`, with `"A"`.
- In part (a), you must check that your solution works if `s` is empty. For the given algorithm, `count < s.size()` will fail and short-circuit the test, which is desirable since `s.findkth(1)`

will violate the precondition of `findkth(k)`—namely, that `k` cannot be greater than `size()`.

- The parameter for `s.findkth` must be greater than 0, hence the use of `s.findkth(count+1)` in part (a).
- For the first solution in part (b), you get a subtle intent error if your last step is `s.remove(s.findkth(i))`. Suppose that `s` is initially `{"FLY", "ASK", "ANT"}`. After the method call `s.remove(s.findkth(1))`, `s` will be `{"FLY", "ASK"}`. After the statement `s.remove(s.findkth(2))`, `s` will be `{"ASK"}`!! The point is that `s` is adjusted after each call to `s.remove`. The algorithm that works is this: If *N* is the number of words that start with "A", simply remove the first element in the list *N* times. Note that the alternative solution avoids the pitfall described by simply repeatedly removing the first element if it starts with "A." The alternative solution, however, has its own pitfall: The algorithm can fail if a test for `s` being empty isn't done for each iteration of the `while` loop.

Scoring Rubric: Word Set

Part (a)	`countA`	**6 points**
+1	use a count variable (declare, initialize, return)	
+1	loop over the word set using `findkth`	
+1	`findkth(count + 1)`	
+1	`substring(0, 1)`	
+1	`equals("A")`	
+1	update `count`	
Part (b)	`removeA`	**3 points**
+1	call to `countA`	
+1	`for` loop	
+1	remove each word that starts with "A"	

2.
```
public class Outfit extends ClothingItem
{
    private Shoes shoes;
    private Pants pants;
    private Top top;

    public Outfit (Shoes aShoes, Pants aPants, Top aTop)
    {
        shoes = aShoes;
        pants = aPants;
        top = aTop;
    }

    public String getDescription()
    {
        return shoes.getDescription() + "/" + pants.getDescription()
                + "/" + top.getDescription() + " outfit";
    }
    public double getPrice()
    {
        if (shoes.getPrice() + pants.getPrice() >= 100
                ||shoes.getPrice() + top.getPrice() >= 100
                ||top.getPrice() + pants.getPrice() >= 100)
            return 0.75 * (shoes.getPrice() + pants.getPrice() +
                    top.getPrice());
        else
            return 0.90 * (shoes.getPrice() + pants.getPrice() +
                    top.getPrice());
    }
}
```

Note

- To access the price and descriptions of items that make up an outfit, your class needs to have variables of type `Shoes`, `Pants`, and `Top`.
- The private instance variables in the `Outfit` class should not be of type `String`! Note that in that case, the ordering of the

parameters—shoes, pants, and top—becomes irrelevant, and your solution violates the specification that a compile-time error occurs with different ordering of the parameters.

Scoring Rubric: Clothing Item

	`Outfit` class	**9 points**
+1	class header with keyword `extends`	
+1	private instance variables of types `Shoes`, `Pants`, and `Top`	
+1	`Outfit` constructor with parameters of types `Shoes`, `Pants`, and `Top`	
+1	assignment of instance variables in constructor	
+1	`getDescription` header	
+1	return concatenated string of outfit descriptions	
+1	`getPrice` header	
+1	test for amount of discount	
+1	return corresponding price with correct discount	

3. (a)

```
public void printNotes()
{
    int count = 1;
    for (Note note: noteList)
    {
        System.out.println(count + ". " + note.getNote());
        count++;
    }
}
```

Alternative solution for part (a):

```
public void printNotes()
{
    for (int index = 0; index < noteList.size(); index++)
        System.out.println(index + 1 + ". "
           + noteList.get(index).getNote());
}
```

(b)

```
public void removeNotes(String str)
{
    int index = 0;
    while (index < noteList.size())
    {
        String note = noteList. get(index).getNote();
        if (note.indexOf(str) == -1)
            index++;
        else
            noteList.remove(index);
    }
}
```

Note

- In part (b), you should increment the index only if you don’t remove a note. This is because removing an element causes all notes following the removed item to shift one slot to the left. If, at the same time, the index moves to the right, you may miss elements that need to be removed.

Scoring Rubric: Note Keeper

Part (a)	`printNotes`	**4 points**
+1	initialize count of notes	
+1	loop over notes in `noteList`	
+1	print current number and corresponding note	
+1	increment count	

Part (b)	removeNotes	5 points
+1	loop over notes in noteList	
+1	get Note that corresponds to current index	
+1	use getNote to access current note as string	
+1	test whether parameter str is contained in current note	
+1	remove note containing str	

4. (a)

```
public boolean twoTogether()
{
    for (int r = 0; r < NUM_ROWS; r++)
        for (int c = 0; c < SEATS_PER_ROW-1; c++)
            if (seats[r][c] == 0 && seats[r][c+1] == 0)
            {
                seats[r][c] = 1;
                seats[r][c+1] = 1;
                return true;
            }
    return false;
}
```

(b)
```
public int findAdjacent(int row, int seatsNeeded)
{
    int index = 0, count = 0, lowIndex = 0;
    while (index < SEATS_PER_ROW)
    {
        while (index < SEATS_PER_ROW && seats[row][index] == 0)
        {
            count++;
            index++;
            if (count == seatsNeeded)
                return lowIndex;
        }
        count = 0;
        index++;
        lowIndex = index;
    }
    return -1;
}
```

Note

- In part (a), you need the test `c < SEATS_PER_ROW-1`, because when you refer to `seats[r][c+1]`, you must worry about going off the end of the row and causing an `ArrayIndexOutOfBounds` exception.
- In part (b), every time you increment `index`, you need to test that it is in range. This is why you need this test twice: `index < SEATS_PER_ROW`.
- In part (b), every time you reset the count, you need to reset the `lowIndex`, because this is the value you're asked to return.
- In parts (a) and (b), the final `return` statements are executed only if all rows in the show have been examined unsuccessfully.

Scoring Rubric: Theater Seats

Part (a)	`twoTogether`	4 points
+1	traverse all seats	
+1	test for two adjacent empty seats	
+1	assign seats as taken	
+1	return `false` if two together not found	
Part (b)	`findAdjacent`	**5 points**
+1	nested loop to search for seats together	
+1	range check for `seats[row][index]`	
+1	update counts for inner and outer loops	
+1	update indexes for inner and outer loops	
+1	test `count` to see if number of seats needed has been found	