a)

Table 4.3 `if/else` Options

Situation	Construct	Basic form

You want to execute any combination of controlled statements	Sequential `if`s	`if (<test1>) {`
		`  <statement1>;`
		`}`
		`if (<test2>) {`
		`  <statement2>;`
		`}`
		`if (<test3>) {`
		`  <statement3>;`
		`}`

You want to execute zero or one of the controlled statements	Nested `if`s ending in test	`if (<test1>) {`
		`  <statement1>;`
		`} else if (<test2>) {`
		`  <statement2>;`
		`} else if (<test3>) {`
		`  <statement3>;`
		`}`

You want to execute exactly one of the controlled statements	Nested `if`s ending in `else`	`if (<test1>) {`
		`  <statement1>;`
		`} else if (<test2>) {`
		`  <statement2>;`
		`} else {`
		`  <statement3>;`
		`}`

b)

Common Programming Error

Choosing the Wrong `if/else` Construct

Suppose that your instructor has told you that grades will be determined as follows:

A for scores ≥ 90
B for scores ≥ 80
C for scores ≥ 70
D for scores ≥ 60
F for scores < 60

You can translate this scale into code as follows:

```
String grade;
if (score >= 90) {
    grade = "A";
}
if (score >= 80) {
    grade = "B";
}
if (score >= 70) {
    grade = "C";
}
if (score >= 60) {
    grade = "D";
}
if (score < 60) {
    grade = "F";
}
```

However, if you then try to use the variable `grade` after this code, you'll get this error from the compiler:

```
variable grade might not have been initialized
```

This is a clue that there is a problem. The Java compiler is saying that it believes there are paths through this code that will leave the variable `grade` uninitialized. In fact, the variable will always be initialized, but the compiler cannot figure this out. We can fix this problem by giving an initial value to `grade`:

```
String grade = "no grade";
```

This change allows the code to compile. But if you compile and run the program, you will find that it gives out only two grades: D and F. Anyone who has a score of at least 60 ends up with a D and anyone with a grade below 60 ends up with an F. And even though the compiler complained that there was a path that would allow `grade` not to be initialized, no one ever gets a grade of "no grade."

The problem here is that you want to execute exactly one of the assignment statements, but when you use sequential `if` statements, it's possible for the program to execute several of them sequentially. For example, if a student has a score of 95, that student's `grade` is set to `"A"`, then reset to `"B"`, then reset to `"C"`, and finally reset to `"D"`. You can fix this problem by using a nested `if/else` construct:

```
String grade;
if (score >= 90) {
    grade = "A";
} else if (score >= 80) {
    grade = "B";
} else if (score >= 70) {
    grade = "C";
} else if (score >= 60) {
    grade = "D";
} else { // score < 60
    grade = "F";
}
```

You don't need to set `grade` to `"no grade"` now because the compiler can see that no matter what path is followed, the variable `grade` will be assigned a value (exactly one of the branches will be executed).

c)

```
1 // Finds the sum of a sequence of numbers.
2
3 import java.util.*;
4
5 public class ExamineNumbers1 {
6     public static void main(String[] args) {
7         System.out.println("This program adds a sequence of");
8         System.out.println("numbers.");
9         System.out.println();
10
11        Scanner console = new Scanner(System.in);
12
13        System.out.print("How many numbers do you have? ");
14        int totalNumber = console.nextInt();
15
16        double sum = 0.0;
17        for (int i = 1; i <= totalNumber; i++) {
18            System.out.print("    #" + i + "? ");
19            double next = console.nextDouble();
20            sum += next;
21        }
22        System.out.println();
23
24        System.out.println("sum = " + sum);
25    }
26 }
```

The program's execution will look something like this (as usual, user input is boldface):

```
This program adds a sequence of
numbers.

How many numbers do you have? 6
    #1? 3.2
    #2? 4.7
    #3? 5.1
    #4? 9.0
    #5? 2.4
    #6? 3.1

sum = 27.5
```

D)

```
1 // Finds the average of a sequence of numbers as well as
2 // reporting how many of the user-specified numbers were negative.
3
4 import java.util.*;
5
6 public class ExamineNumbers2 {
7     public static void main(String[] args) {
8         System.out.println("This program examines a sequence");
9         System.out.println("of numbers to find the average");
10        System.out.println("and count how many are negative.");
11        System.out.println();
12
13        Scanner console = new Scanner(System.in);
14
15        System.out.print("How many numbers do you have? ");
16        int totalNumber = console.nextInt();
17
18        int negatives = 0;
19        double sum = 0.0;
20        for (int i = 1; i <= totalNumber; i++) {
21            System.out.print("    #" + i + "? ");
22            double next = console.nextDouble();
23            sum += next;
24            if (next < 0) {
25                negatives++;
26            }
27        }
28        System.out.println();
29
30        if (totalNumber <= 0) {
31            System.out.println("No numbers to average");
32        } else {
33            double average = sum / totalNumber;
34            System.out.println("average = " + average);
35        }
36        System.out.println("# of negatives = " + negatives);
37    }
38 }
```

```
This program examines a sequence
of numbers to find the average
and count how many are negative.

How many numbers do you have? 8
    #1? 2.5
    #2? 9.2
    #3? -19.4
    #4? 208.2
    #5? 42.3
    #6? 92.7
    #7? -17.4
    #8? 8

average = 40.7625
# of negatives = 2
```

E)

```
1 public class Roundoff {
2     public static void main(String[] args) {
3         double n = 1.0;
4         for (int i = 1; i <= 10; i++) {
5             n += 0.1;
6             System.out.println(n);
7         }
8     }
9 }
```

This program presents a classic cumulative sum with a loop that adds 0.1 to the number `n` each time the loop executes. We start with `n` equal to `1.0` and the loop iterates 10 times, which we might expect to print the numbers 1.1, 1.2, 1.3, and so on through 2.0. Instead, it produces the following output:

```
1.1
1.2000000000000002
1.3000000000000003
1.4000000000000004
1.5000000000000004
1.6000000000000005
1.7000000000000006
1.8000000000000007
1.9000000000000008
2.000000000000001
```

F)

Table 4.4 Differences between `char` and `String`

	`char`	`String`
Type of value	primitive	object
Memory usage	2 bytes	depends on length
Methods	none	`length`, `toUpperCase`, . . .
Number of letters	exactly 1	0 to many
Surrounded by	apostrophes: `'c'`	quotes: `"Str"`
Comparing	<, >=, ==, . . .	`equals`

G)

```
public static int count(String text, char c) {
    int found = 0;
    for (int i = 0; i < text.length(); i++) {
        if (text.charAt(i) == c) {
            found++;
        }
    }
    return found;
}
```

H)

Table 4.5 Useful Methods of the `Character` Class

Method	Description	Example
`getNumericValue(ch)`	Converts a character that looks like a number into that number	`Character.getNumericValue('6')` returns `6`
`isDigit(ch)`	Whether or not the character is one of the digits `'0'` through `'9'`	`Character.isDigit('X')` returns `false`
`isLetter(ch)`	Whether or not the character is in the range `'a'` to `'z'` or `'A'` to `'Z'`	`Character.isLetter('f')` returns `true`
`isLowerCase(ch)`	Whether or not the character is a lowercase letter	`Character.isLowerCase('Q')` returns `false`
`isUpperCase(ch)`	Whether or not the character is an uppercase letter	`Character.isUpperCase('Q')` returns `true`
`toLowerCase(ch)`	The lowercase version of the given letter	`Character.toLowerCase('Q')` returns `'q'`
`toUpperCase(ch)`	The uppercase version of the given letter	`Character.toUpperCase('x')` returns `'X'`

I)

Table 4.6 Common Format Specifiers

Specifier	Result
`%d`	Integer
`%8d`	Integer, right-aligned, 8-space-wide field
`%-6d`	Integer, left-aligned, 6-space-wide field
`%f`	Floating-point number
`%12f`	Floating-point number, right-aligned, 12-space-wide field
`%.2f`	Floating-point number, rounded to nearest hundredth
`%16.3f`	Floating-point number, rounded to nearest thousandth, 16-space-wide field
`%s`	String
`%8s`	String, right-aligned, 8-space-wide field
`%-9s`	String, left-aligned, 9-space-wide field

J)

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {

        System.out.printf("%5d", i * j);
    }
    System.out.println();
}
```

This code produces the following output:

```
    1    2    3    4    5    6    7    8    9   10
    2    4    6    8   10   12   14   16   18   20
    3    6    9   12   15   18   21   24   27   30
    4    8   12   16   20   24   28   32   36   40
    5   10   15   20   25   30   35   40   45   50
    6   12   18   24   30   36   42   48   54   60
    7   14   21   28   35   42   49   56   63   70
    8   16   24   32   40   48   56   64   72   80
    9   18   27   36   45   54   63   72   81   90
   10   20   30   40   50   60   70   80   90  100
```