Building Java Programs 6th edition Reges Test Bank

SKU: 9780135429914-TEST-BANK

Sample Final Exam #6

(Summer 2008; thanks to Hélène Martin)

1. Array Mystery

Consider the following method:

```
public static void arrayMystery(String[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] + a[a.length - 1 - i];
    }
}
```

Indicate in the right-hand column what values would be stored in the array after the method `arrayMystery` executes if the array in the left-hand column is passed as a parameter to it.

Original Contents of Array	Final Contents of Array
`String[] a1 = {"a", "b", "c"};` `arrayMystery(a1);`	_____
`String[] a2 = {"a", "bb", "c", "dd"};` `arrayMystery(a2);`	_____
`String[] a3 = {"z", "y", "142", "w", "xx"};` `arrayMystery(a3);`	_____

Building Java Programs 6th edition Reges Test Bank

2. Reference Semantics Mystery

The following program produces 4 lines of output. Write the output below, as it would appear on the console.

```
public class Pokemon {
    int level;

    public Pokemon(int level) {
        this.level = level;
    }
}

public class ReferenceMystery {
    public static void main(String[] args) {
        int hp = 10;
        Pokemon squirtle = new Pokemon(5);

        battle(squirtle, hp);
        System.out.println("Level " + squirtle.level + ", " + hp + " hp");

        hp = hp + squirtle.level;

        battle(squirtle, hp + 1);
        System.out.println("Level " + squirtle.level + ", " + hp + " hp");
    }

    public static void battle(Pokemon poke, int hp) {
        poke.level++;
        hp -= 5;
        System.out.println("Level " + poke.level + ", " + hp + " hp");
    }
}
```


Building Java Programs 6th edition Reges Test Bank

3. Inheritance Mystery

Assume that the following classes have been defined:

```
public class Dog extends Cat {
    public void m1() {
        m2();
        System.out.print("dog 1    ");
    }
}

public class Lion extends Dog {
    public void m2() {
        System.out.print("lion 2    ");
        super.m2();
    }

    public String toString() {
        return "lion";
    }
}
```

```
public class Cat {
    public void m1() {
        System.out.print("cat 1    ");
    }

    public void m2() {
        System.out.print("cat 2    ");
    }

    public String toString() {
        return "cat";
    }
}
```

Given the classes above, what output is produced by the following code?

```
Cat[] elements = {new Dog(), new Cat(), new Lion()};
for (int i = 0; i < elements.length; i++) {
    elements[i].m1();
    System.out.println();
    elements[i].m2();
    System.out.println();
    System.out.println(elements[i]);
    System.out.println();
}
```


Building Java Programs 6th edition Reges Test Bank

4. File Processing

Write a static method `evaluate` that accepts as a parameter a `Scanner` containing a series of tokens representing a numeric expression involving addition and subtraction and that returns the value of the expression. For example, if a `Scanner` called `data` contains the following tokens:

```
4.2 + 3.4 - 4.1
```

The call of `evaluate(data);` should evaluate the result as (4.2+3.4-4.1) = (7.6-4.1) = **3.5** and should return this value as its result. Every expression will begin with a real number and then will have a series of operator/number pairs that follow. The operators will be either + (addition) or - (subtraction). As in the example above, there will be spaces separating numbers and operators. You may assume the expression is legal.

Your program should evaluate operators sequentially from left to right. For example, for this expression:

```
7.3 - 4.1 - 2.0
```

your method should evaluate the operators as follows:

```
7.3 - 4.1 - 2.0 = (7.3 - 4.1) - 2.0 = 3.2 - 2.0 = 1.2
```

The `Scanner` might contain just a number, in which case your method should return that number as its result.

Building Java Programs 6th edition Reges Test Bank

5. File Processing

Write a static method `blackjack` that accepts as its parameter a `Scanner` for an input file containing a hand of playing cards, and returns the point value of the hand in the card game Blackjack.

A card has a rank and a suit. There are 13 ranks: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. There are 4 suits: Clubs, Diamonds, Hearts, and Spades. A Blackjack hand's point value is the sum of its cards' point values. A card's point value comes from its rank; the suit is irrelevant. In this problem, cards are worth the following points:

Rank	Point Value
2-10	The card's rank (for example, a 7 is worth 7 points)
Jack (`J`), Queen (`Q`), King (`K`)	10 points each
Ace (`A`)	11 points *(for this problem; simplified compared to real Blackjack)*

The input file contains a single hand of cards, each represented by a pair of "*<rank>* *<suit>*" tokens. For example:

```
5 Diamonds
Q Spades
2 Spades  3 Hearts
```

Given the above input, your method should return 20, since the cards' point values are 5 + 10 + 2 + 3 = 20.

The input can be in mixed casing, have odd spacing between tokens, and can be split across lines. For example:

```
2 Hearts
  j SPADES   a  Diamonds
2 ClUbS
   A
hearts
```

Given the above input, your method should return 36, since the cards' point values are 2 + 10 + 11 + 2 + 11 = 36.

You may assume that the `Scanner` contains at least 1 card (two tokens) of input, and that no line will contain any tokens other than valid card data. The real game of Blackjack has many other rules that you should ignore for this problem, such as the notion of going "bust" once you exceed a score of 21.

Building Java Programs 6th edition Reges Test Bank

6. Array Programming

Write a static method named `allPlural` that accepts an array of strings as a parameter and returns `true` only if every string in the array is a plural word, and `false` otherwise. For this problem a plural word is defined as any string that ends with the letter S, case-insensitively. The empty string `""` is *not* considered a plural word, but the single-letter string "s" or "S" is. Your method should return `true` if passed an empty array (one with 0 elements).

The table below shows calls to your method and the expected values returned:

Array	Call and Value Returned
`String[] a1 = {"snails", "DOGS", "Cats"};`	`allPlural(a1)` returns `true`
`String[] a2 = {"builds", "Is", "S", "THRILLs", "CS"};`	`allPlural(a2)` returns `true`
`String[] a3 = {};`	`allPlural(a3)` returns `true`
`String[] a4 = {"She", "sells", "sea", "SHELLS"};`	`allPlural(a4)` returns `false`
`String[] a5 = {"HANDS", "feet", "toes", "OxEn"};`	`allPlural(a5)` returns `false`
`String[] a6 = {"shoes", "", "socks"};`	`allPlural(a6)` returns `false`

For full credit, your method should not modify the array's elements.

Building Java Programs 6th edition Reges Test Bank

7. Array Programming

Write a static method named `reverseChunks` that accepts two parameters, an array of integers *a* and an integer "chunk" size *s*, and reverses every *s* elements of *a*. For example, if *s* is 2 and array *a* stores `{1, 2, 3, 4, 5, 6}`, *a* is rearranged to store `{2, 1, 4, 3, 6, 5}`. With an *s* of 3 and the same elements `{1, 2, 3, 4, 5, 6}`, array *a* is rearranged to store `{3, 2, 1, 6, 5, 4}`. The chunks on this page are underlined for convenience.

If *a*'s length is not evenly divisible by *s*, the remaining elements are untouched. For example, if *s* is 4 and array *a* stores `{5, 4, 9, 2, 1, 7, 8, 6, 2, 10}`, *a* is rearranged to store `{2, 9, 4, 5, 6, 8, 7, 1, 2, 10}`. It is also possible that *s* is larger than *a*'s entire length, in which case the array is not modified at all. You may assume that *s* is 1 or greater (an *s* of 1 would not modify the array). If array *a* is empty, its contents should remain unchanged.

The following table shows some calls to your method and their expected results:

Array and Call	Array Contents After Call
`int[] a1 = {20, 10, 30, 60, 50, 40};` `reverseChunks(a1, 2);`	`{10, 20, 60, 30, 40, 50}`
`int[] a2 = {2, 4, 6, 8, 10, 12, 14, 16};` `reverseChunks(a2, 3);`	`{6, 4, 2, 12, 10, 8, 14, 16}`
`int[] a3 = {7, 1, 3, 5, 9, 8, 2, 6, 4, 10, 0, 12};` `reverseChunks(a3, 5);`	`{9, 5, 3, 1, 7, 10, 4, 6, 2, 8, 0, 12}`
`int[] a4 = {1, 2, 3, 4, 5, 6};` `reverseChunks(a4, 8);`	`{1, 2, 3, 4, 5, 6}`
`int[] a5 = {};` `reverseChunks(a5, 2);`	`{}`

Building Java Programs 6th edition Reges Test Bank

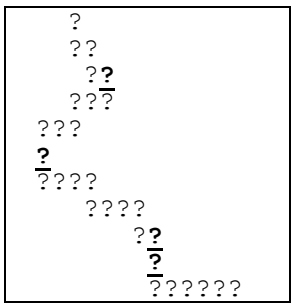
8. Critters

Write a class `Minnow` that extends `Critter` from HW8, along with its movement and eating behavior. All other aspects of `Minnow` use the defaults. Add fields, constructors, etc. as necessary to your class.

`Minnow` objects initially move in a S/E/S/E/... pattern. However, when a `Minnow` encounters food (when its `eat` method is called), it should do all of the following:

- **Do *not* eat the food.**
- **Start the movement cycle over.** In other words, the next move after `eat` is called should always be South.
- **Lengthen and reverse the horizontal portion of the movement cycle pattern.**
  The `Minnow` should reverse its horizontal direction and increase its horizontal movement distance by 1 for subsequent cycles. For example, if the `Minnow` had been moving S/E/S/E, it will now move S/**W/W**/S/**W/W**. If it hits a second piece of food, it will move S/**E/E/E**/S/**E/E/E**, and a third, S/**W/W/W/W**/S/**W/W/W/W**, and so on.

The following is an example timeline of a particular `Minnow` object's movement. The timeline below is also drawn in the diagram at right. Underlined occurrences mark squares where the `Minnow` found food.



- S, E, S, E (hits food)
- S, W, W, S, W, W, S (hits food)
- S, E, E, E, S, E, E, E, S, E (hits food)
- S (hits food)
- S, E, E, E, E, E, S, E, E, E, E, E, ...

Building Java Programs 6th edition Reges Test Bank

9. Classes and Objects

Suppose that you are provided with a pre-written class `Date` as described at right. (The headings are shown, but not the method bodies, to save space.) Assume that the fields, constructor, and methods shown are already implemented. You may refer to them or use them in solving this problem if necessary.

Write an instance method named **`bound`** that will be placed inside the `Date` class to become a part of each `Date` object's behavior. The `bound` method constrains a `Date` to within a given range of dates. It accepts two other `Date` objects *d1* and *d2* as parameters; *d1*'s date is guaranteed to represent a date that comes no later in the year than *d2*'s date.

The `bound` method makes sure that this `Date` object is between *d1*'s and *d2*'s dates, inclusive. If this `Date` object is not between those dates inclusive, it is adjusted to the nearest date in the acceptable range. The method returns a result of `true` if this `Date` was within the acceptable range, or `false` if it was shifted.

For example, given the following `Date` objects:

```
Date date1 = new Date(7, 12);
Date date2 = new Date(10, 31);
Date date3 = new Date(9, 19);
Date bound1 = new Date(8, 4);
Date bound2 = new Date(9, 26);
Date bound3 = new Date(12, 25);
```

The following calls to your method should adjust the given `Date` objects to represent the following dates and should return the following results:

call	date becomes	returns
`date1.bound(bound1, bound2)`	`8/4`	`false`
`date2.bound(bound1, bound2)`	`9/26`	`false`
`date3.bound(bound1, bound3)`	`9/19`	`true`
`date2.bound(bound3, bound3)`	`12/25`	`false`

```
// Each Date object stores a single
// month/day such as September 19.
// This class ignores leap years.

public class Date {
    private int month;
    private int day;

    // Constructs a date with
    // the given month and day.
    public Date(int m, int d)

    // Returns the date's day.
    public int getDay()

    // Returns the date's month.
    public int getMonth()

    // Returns the number of days
    // in this date's month.
    public int daysInMonth()

    // Modifies this date's state
    // so that it has moved forward
    // in time by 1 day, wrapping
    // around into the next month
    // or year if necessary.
    // example:  9/19 -> 9/20
    // example:  9/30 -> 10/1
    // example: 12/31 -> 1/1
    public void nextDay()


    // your method would go here

}
```


Building Java Programs 6th edition Reges Test Bank