

Java exercises - solutions

Please see the separate sheet about running Eclipse if you are not familiar with the Eclipse IDE.

Task 1

Write a Java method to compute and print the list of prime numbers less than a given parameter. The method signature should be as follows:

void printPrimes (int max)

```
void printPrimes(int max) {
    for (int n = 2; n < max; n++) {
        boolean prime = true;
        for (int f = 2; f <= Math.sqrt(n); f++) {
            if (n % f == 0) {
                prime = false;
                break;
            }
        }
        if (prime) {
            System.out.println(n + " is prime");
        }
    }
}
```

Task 2

Write a Java method to compute and return the Nth Fibonacci number, which is a sequence of numbers where each number is the sum of the preceding two (see https://en.wikipedia.org/wiki/Fibonacci_number if you can't remember the details). The method signature should be as follows:

int computeFibonacci (int n)

```
int computeFibonacci(int n) {
    int result = 1, lastResult = 1;
    for (int i = 2; i < n; i++) {
        int temp = result;
        result = result + lastResult;
        lastResult = temp;
    }
    return result;
}
```

}

Task 3

Implement a method **computeScore** that correctly computes and returns the score that you would receive in the game of Scrabble¹ for the string given as a parameter – that is, you must add up the score of each letter in the string and return the total value. Refer to the following table of values for each letter:

A: 1	B: 3	C: 3	D: 2	E: 1
F: 4	G: 2	H: 4	I: 1	J: 8
K: 5	L: 1	M: 3	N: 1	O: 1
P: 3	Q: 10	R: 1	S: 1	T: 1
U: 1	V: 4	W: 4	X: 8	Y: 4
Z: 10				

If you encounter any characters other than those 26 letters, then they should not affect the total score, but **you should be sure to consider both upper case and lower case letters**.

The signature for the method must be as follows:

int computeScore (String word)

¹ <https://en.wikipedia.org/wiki/Scrabble> or <http://www.scrabble.com/>

```

int computeScore (String str) {
    // Initialise counter and normalise string
    int total = 0;
    str = str.toLowerCase();

    // Go through the string and add up the score
    for (char c : str.toCharArray()) {
        switch (c) {
            case 'a':
            case 'e':
            case 'i':
            case 'l':
            case 'n':
            case 'o':
            case 'r':
            case 's':
            case 't':
            case 'u':
                total++;
                break;

            case 'd':
            case 'g':
                total += 2;
                break;

            case 'b':
            case 'c':
            case 'm':
            case 'p':
                total += 3;
                break;

            case 'f':
            case 'h':
            case 'v':
            case 'w':
            case 'y':
                total += 4;
                break;

            case 'k':
                total += 5;
                break;

            case 'j':
            case 'x':
                total += 8;
                break;

            case 'q':
            case 'z':
                total += 10;
                break;

            default:
                // Nothing needed here -- we ignore all other
                characters
        }
    }
}

```

```

        }
    }
    // This is the total score
    return total;
}

```

Task 4

Design and implement a class **Monster** representing a (simplified) monster in a monster-battling game. A monster includes a **type** (a String), a number of **hit points** (an int), a number of **attack points** (an int), as well as list of **weaknesses** (i.e., types against which the monster is particularly weak in battle, represented as a String[]).

Define a **Monster** class with appropriate fields with correct access modifiers, as well as getter methods for all fields and an appropriate constructor.

You should also define the following two methods which are used when monsters battle each other:

- `public boolean isWeakAgainst (String otherType)`
 - This method should return **true** if **otherType** is included in this Monster's **weaknesses** list, and **false** if it is not. Don't forget to use **.equals()** to compare String values instead of **"=="**
- `public void removeHitPoints (int pointsToRemove)`
 - Removes the indicated number of hit points from the current monster. If the hit point value becomes negative, then it should be set to zero.
- `public boolean attack (Monster otherMonster)`
 - This method is called when the current monster attacks the other monster. It should proceed as follows:
 - If **otherMonster** is actually this monster (use **"=="** to check), return false immediately
 - If either the current monster or the other monster is knocked out (i.e., hit points == 0), return false immediately
 - Otherwise, check if the other monster is weak against the current monster's type (use **isWeakAgainst()**)
 - If it is not weak, remove only the current monster's attack points from the other monster's hit points (use **removeHitPoints**)
 - If the other monster is weak, remove the attack points plus an additional 20 points (again, use **removeHitPoints**)

```

import java.util.Arrays;

public class Monster {

    protected String type;
    protected int hitPoints;
    protected int attackPoints;
    protected String[] weaknesses;

    public Monster(String type, int hitPoints, int attackPoints,
String[] weaknesses) {
        this.type = type;
        this.hitPoints = hitPoints;
        this.attackPoints = attackPoints;
        this.weaknesses = weaknesses;
    }

    // Getters and setters
    public int getHitPoints() {
        return hitPoints;
    }

    public int getAttackPoints() {
        return attackPoints;
    }

    public String getType() {
        return this.type;
    }

    public boolean attack(Monster otherMonster) {
        // A monster cannot attack itself
        if (otherMonster == this) {
            return false;
        }

        // A monster cannot attack or be attacked if it is knocked
out
        if (this.hitPoints <= 0 || otherMonster.getHitPoints() <=
0) {
            return false;
        }

        // Check if the other monster is weak against our type
        boolean otherIsWeak = otherMonster.isWeakAgainst(type);
        int pointsToRemove = (otherIsWeak) ? this.attackPoints + 20
: this.attackPoints;
        otherMonster.removeHitPoints(pointsToRemove);
        return true;
    }

    private boolean isWeakAgainst(String otherType) {
        for (String weakness : this.weaknesses) {
            if (weakness.equals(otherType)) {
                return true;
            }
        }
        return false;
    }
}

```

```

    }

    public void removeHitPoints(int points) {
        this.hitPoints -= points;
        if (hitPoints <= 0) {
            // Monster is knocked out
            hitPoints = 0;
        }
    }

    @Override
    public String toString() {
        return "Monster [type=" + type + ", hitPoints=" + hitPoints
+ ", attackPoints=" + attackPoints + ", weaknesses="
+ Arrays.toString(weaknesses) + "];"
    }

    public static void main(String[] args) {
        Monster fireMonster = new Monster("Fire", 200, 100, new
String[] { "Water" });
        Monster waterMonster = new Monster("Water", 130, 50, new
String[] { "Fire", "Electric" });
        waterMonster.attack(fireMonster); // Should return true
        System.out.println(fireMonster.getHitPoints()); // Should
print
        fireMonster.getHitPoints();
    }
}

```

Task 5

Based on the **Monster** class created above, you should **refactor** it into a set of classes that are able to represent every type of monster as its own class, instead of using the **type** field to distinguish them. The information that is common to all monsters will be retained in the **Monster** class, which will be made **abstract**, while other information that is relevant only to one of the monster types will be put into the appropriate subclass. The subtypes should be **WaterMonster**, **FireMonster**, and **ElectricMonster**, with the following properties:

- A **FireMonster** has type "Fire" and is weak against Water
- A **WaterMonster** has type "Water" and is weak against Fire and Electric
- An **ElectricMonster** has type "Electric" and has no weaknesses

dodge()

Add an abstract **dodge()** method to the parent **Monster** class – this method should return a **boolean** value and will be implemented in the subclasses to implement the modified **attack()** behaviour described below. The method should have a **protected** access modifier.

The required behaviour for **dodge()** in each subclass is as follows – you should add any necessary fields to each subclass to implement this behaviour:

- **FireMonster**: this method should alternatively return **true** and **false** – that is, the first time it is called, it should return **true**, the next call should return **false**, and so on
- **WaterMonster**: this method should return **true** if the monster's hit points are at least 100, and **false** if they are less than 100.
- **ElectricMonster**: this method should always return **false** – that is, an electric monster should never dodge when attacked.

attack()

The final piece of refactoring is to modify the **attack()** method of **Monster** to use **dodge()** as follows:

- First, call **dodge()** on the monster being attacked.
- If the result is **false**, the attack behaviour as before is implemented.
- If the result is **true**, no hit points are removed from the monster being attacked, but 10 hit points are removed from the monster doing the attacking. The same rules apply here – if the monster's HP goes below zero, then it should be set to zero.

```

import java.util.Arrays;

public abstract class Monster {

    protected String type;
    protected int hitPoints;
    protected int attackPoints;
    protected String[] weaknesses;

    public Monster(String type, int hitPoints, int attackPoints, String[]
weaknesses) {
        this.type = type;
        this.hitPoints = hitPoints;
        this.attackPoints = attackPoints;
        this.weaknesses = weaknesses;
    }

    // Getters and setters
    public int getHitPoints() {
        return hitPoints;
    }

    public int getAttackPoints() {
        return attackPoints;
    }

    public String getType() {
        return this.type;
    }

    public boolean attack(Monster otherMonster) throws MonsterException {
        // A monster cannot attack itself
        if (otherMonster == this) {
            return false;
        }

        // A monster cannot attack or be attacked if it is knocked out
        if (this.hitPoints <= 0 || otherMonster.getHitPoints() <= 0) {
            return false;
        }

        if (otherMonster.dodge()) {
            this.removeHitPoints(10);
            return false;
        } else {
            // Check if the other monster is weak against our type
            boolean otherIsWeak = otherMonster.isWeakAgainst(type);
            int pointsToRemove = (otherIsWeak) ? this.attackPoints + 20
: this.attackPoints;
            otherMonster.removeHitPoints(pointsToRemove);
            return true;
        }
    }

    public abstract boolean dodge();

    public boolean isWeakAgainst(String otherType) {
        for (String weakness : this.weaknesses) {
            if (weakness.equals(otherType)) {
                return true;
            }
        }
        return false;
    }
}

```



```

        private void removeHitPoints(int points) {
            this.hitPoints -= points;
            if (hitPoints <= 0) {
                // Monster is knocked out
                hitPoints = 0;
            }
        }

        @Override
        public String toString() {
            return "Monster [type=" + type + ", hitPoints=" + hitPoints + ",
attackPoints=" + attackPoints + ", weaknesses="
                + Arrays.toString(weaknesses) + "]";
        }
    }

}

public class WaterMonster extends Monster {

    public WaterMonster(int hitPoints, int attackPoints) {
        super("Water", hitPoints, attackPoints, new String[] { "Fire",
"Electric" } );
    }

    @Override
    public boolean dodge() {
        return (hitPoints >= 100);
    }

}

public class FireMonster extends Monster {

    private boolean lastDodge = false;

    public FireMonster(int hitPoints, int attackPoints) {
        super("Fire", hitPoints, attackPoints, new String[] { "Water" });
    }

    @Override
    public boolean dodge() {
        return (lastDodge = !lastDodge);
    }

}

public class ElectricMonster extends Monster {

    public ElectricMonster(int hitPoints, int attackPoints) {
        super("Electric", hitPoints, attackPoints, new String[0]);
    }

    @Override
    public boolean dodge() {
        return false;
    }

}

```

Task 6

Your task is to write a program to check whether a given credit card number is valid or invalid. You can read more about the process of checking credit card numbers at several websites, including <http://www.validcreditcardnumber.com/>; the description below is a summary of the properties that we will be using for this lab.

The first several digits of the card number indicate the issuing network of the card:

- **Visa** card numbers all start with **4**
- **American Express** card numbers all start with **34** or **37**
- **MasterCard** card numbers all start with a number between **51 and 55 (inclusive)**, or with a number between **2221 and 2720 (inclusive)**

The remainder of the card number is allocated by the card issuer. In general, a valid credit card number can be anywhere between 13 and 19 digits; however, depending on the issuer, the valid lengths can vary:

- All **Visa** card numbers are of length **13, 16 or 19**
- All **American Express** card numbers are of length **15**
- All **MasterCard** card numbers are of length **16**

Finally, the final digit of all credit card numbers is a **check digit** – that is, a digit computed from the other digits of the card number. Including this digit means that simple errors in manually entering a card number (such as a single mistyped digit or permutations of successive digits) will be caught because the card will not be validated. For credit cards, the check digit is computed using the Luhn Algorithm (https://en.wikipedia.org/wiki/Luhn_algorithm).

To test whether a credit card number is valid, proceed as follows (adapted from the Wikipedia page above):

1. **Proceeding from the rightmost digit and moving left**, double the value of **every second digit**. If the result of this doubling is greater than 9, then subtract 9 from the product.
2. Take the sum of the resulting digits.
3. The credit card number is valid if and only if the sum is a multiple of 10.

As a concrete example, consider the number “79927398713”. The validation process would proceed as shown in the following table.

<i>Account number</i>	7	9	9	2	7	3	9	8	7	1	3
<i>Double every other</i>	7	18	9	4	7	6	9	16	7	2	3
<i>Digits to sum</i>	7	9	9	4	7	6	9	7	7	2	3

The sum of the resulting digits is $7+9+9+4+7+6+9+7+7+2+3=70$, so the number is valid according to the Luhn algorithm.

What you need to do

You must implement a method to validate a credit card number. The method signature should be as follows:

boolean checkCardNumber (String cardNumber)

Your method should carry out all of the checks listed on the previous page and should return **true** if the card number is valid and **false** if it is not. You can use the sample (fictional) credit card numbers from websites such as <http://www.getcreditcardnumbers.com/> to test your code. Be sure to test on both valid and invalid numbers!

```
boolean checkCardNumber (String cardNumber) {
    // Check length first -- can't be valid if length is out of range
    if (cardNumber.length() < 13 || cardNumber.length() > 19) {
        return false;
    }

    // Check for a valid prefix, and also check for correct length while
    doing this
    switch (cardNumber.charAt(0)) {
        case '4':
            // Visa cards should be 13, 16, or 19 digits long
            if (cardNumber.length() != 13 && cardNumber.length() != 16
&& cardNumber.length() != 19) {
                return false;
            }
            break;

        case '3':
            // American Express cards should have 4 or 7 as second
            digit and should be 15 digits long
            if (cardNumber.length() != 15) {
                return false;
            }
            char digit2 = cardNumber.charAt(1);
            if (digit2 != '4' && digit2 != '7') {
                return false;
            }
            break;

        case '5':
            // MasterCard (old cards) -- should 16 digits with prefix
            between 51 and 55
            if (cardNumber.length() != 16) {
                return false;
            }
            int prefix = Integer.valueOf(cardNumber.substring(0, 2));
            if (prefix < 51 || prefix > 55) {
                return false;
            }
            break;

        case '2':
            // MasterCard (new cards) -- should be 16 digits with
            prefix between 2221 and 2720
            if (cardNumber.length() != 16) {
                return false;
            }
            int prefix2 = Integer.valueOf(cardNumber.substring(0, 4));
            if (prefix2 < 2221 || prefix2 > 2720) {
                return false;
            }
            break;

        default:
            // Prefix must be invalid so can't be a valid number
    }
```

```

        return false;
    }

    // If we get here, prefix and length must be right. Time to do the Luhn
check.
    boolean even = false;
    int total = 0;

    // Loop backwards through the string one character at a time
    for (int i = cardNumber.length() - 1; i >= 0; i--) {
        // Convert to a number
        int digit = Character.getNumericValue(cardNumber.charAt(i));

        // Double even digits
        if (even) {
            int sum = digit * 2;
            if (sum > 9) {
                sum -= 9;
            }
            total += sum;
        } else {
            total += digit;
        }

        // Even and odd alternate
        even = !even;
    }

    // Card is valid exactly if final total is a multiple of 10
    return (total % 10 == 0);
}

```