Linnéuniversitetet
Linnaeus University

Problem Solving and Programming

Assignment 3
In the third assignment we will begin to create real, object oriented solutions to our tasks. As Java is an object oriented language, it is best used for creating programs that use the object oriented concepts. The lectures talk about object orientation being more of a mind set than a notation and therefore it is important to solve these tasks *in an object oriented fashion*, it might not be enough just to present a working solution.

Carefully read the assignment rules as defined on Moodle. In brief, do the assignments yourself, hand them in at the deadline and don't cheat.

Problems? Do not hesitate to ask your teaching assistant at the practical meetings (or the teacher at the lectures) if you have any problems. You can also post a question in the assignment forum in Moodle.

Validating input You only have to validate the input if the task explicitly says you have to. If nothing is mentioned, you can assume that the user will input the correct type of value.

Submission We are only interested in your .java files. Hence, zip the directory named `YourLnuUserName_assign3` (inside directory named `src`) and submit it using the Moodle submission system.

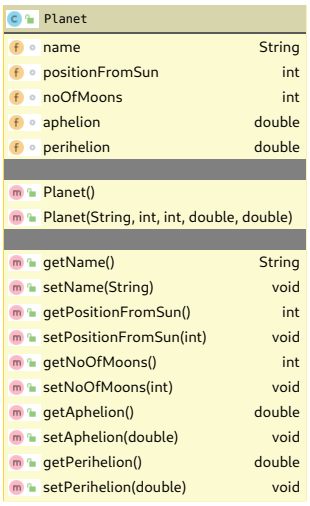
3 Foundation of Object Orientation

The following tasks will help you understand how to build object oriented programs in Java. Notice that *each* class must be a *separate* file – if you fail to do this, you will fail the task. Each task needs a separate class that has a `main` method from which the class, or classes, in the task are instantiated.

Also notice that only 3.7 and 3.8 are VG tasks, which means that task 3.9 is mandatory for everyone.

3.1 Planet

In this first task, we are going to create a class for planets. The class can be seen in UML to the right, with fields (attributes), constructors and methods. As seen, the information we need to store about a planet is its name, the number from the sun, the number of moons as well as aphelion and perihelion – last two being the farthest and the closest the planet is to the sun. This is measured in AU, astronomical units.



Create the class `Planet` in the file `Planet.java`. As seen, a number of methods must be constructed, and it is up to you to see too that they ensure that the fields are protected (encapsulated). For the name the methods should make sure that an empty string isn't sent (if it is, the name can be set to "Unknown planet"). When it comes to the position from the sun, a valid number is between 1 and 9 (yes, we know that Pluto isn't a planet any more, but we feel sad for it and will make room for it). If an invalid number is sent, the value should be set to 0. The number of moons should be a number between 0 and 100, and set to 0 if an invalid number is sent. Aphelion and perihelion are values between 0 and 50 or 0 and 30 respec-

tively – again set the value to 0 if an invalid value is sent. The constructor that takes parameters should use these methods to set the values.

Also create another file called `MarsMain.java` which will be used to instantiate the objects of the class just created. In this class you should create the planet Mars and give it the appropriate information. It is up to you to choose either to call the individual methods to fill in the data, or to use the constructor which accepts parameters. The program should then print out the information about Mars in a similar fashion to below:

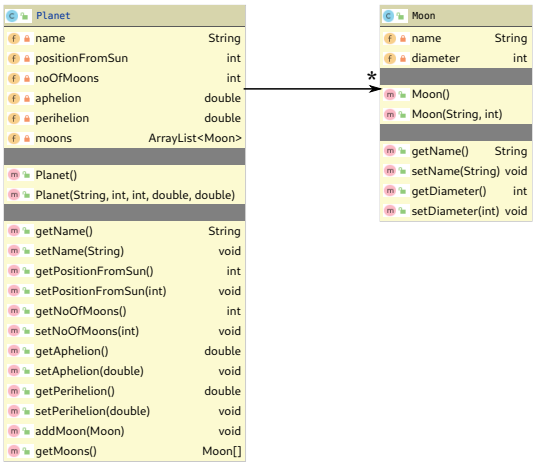
```
Mars:
    Position:   4
    Moons:      2
    Aphelion:   1.666
    Perihelion: 1.382
```

3.2 Moon

Now create a class for a moon. It should contain name of the moon and the diameter of it (in kilometres, not miles). The attributes (fields) should be well encapsulated with similar rules as for the Planet. Again, this is a separate file, called `Moon.java` and you need to create an additional file with a `main` method, call it `MoonMain.java`. In this file, instantiate an example moon and print it out. An example can be seen below:

```
Largest moon in the solar system is Ganymede which
is 5262 km in diameter.
```

3.3 Planet with moons



Many of the planets that we know of have moons and in this task you will change the `Planet` class so that it can handle a number of moons. The relation can be seen in the image to the right. In this case it is a *unidirectional* association from `Planet` to `Moon` meaning that the moon *does not* need to know of which planet it belongs. Also notice that each planet can have *a number of* moons. To be able to do this, you need to add two methods and one attribute as shown in the image. Notice that the methods *have to have* the exact signature as shown here, you may not change anything in it. You must also see to that encapsulation holds, that is, that nothing can change the content of the `ArrayList` itself (except, of course, of the method `add()`).

Mark any and all changes that you do in the class `Planet` with comments so that it is clearly visible where they are.

Also create a new main program called `MoonsMain.java` that instantiate the planet Mars (again) and adds the two moons Phobos and Deimos. The main program should also print out the name of the planet as well as the names of the moons, see below:

```
Moons of Mars:
    Phobos
    Deimos
```

3.4 Stone planets

The four inner planets in our solar system are also known as the *stone planets*. Create a program called `StonePlanets.java` that creates an array of four planets and populates it with the four stone planets as well as the moons that some of them have. The program should then print out the information that can be seen below.

```
Planet Mercury has aphelion 0.47 AU, perihelion 0.31 AU and 0 moons.
Planet Venus has aphelion 0.72 AU, perihelion 0.72 AU and 0 moons.
Planet Earth has aphelion 1.0 AU, perihelion 0.99 AU and 1 moons.
    The Moon
Planet Mars has aphelion 1.666 AU, perihelion 1.382 AU and 2 moons.
    Phobos
    Deimos
```

Notice that you must not make any changes in the previous classes so the check if the moons should be printed or not must be done in the main program (`StonePlanets.java`) and you have to create moons and add them in this file as well. When printing, you must be using an iteration to go through the array of planets (and another to go through the moons).

3.5 Points class

Create a class `Point` that represents a point on the format (x, y). Deduce how it should be implemented by looking at the following code:

```
Point p1 = new Point();
Point p2 = new Point(3,4);

System.out.println(p1.toString());   // ==> (0,0)
System.out.println(p2.toString());   // ==> (3,4)

if (p1.isEqualTo(p2))              // False!
    System.out.println("The two points are equal");

double dist = p1.distanceTo(p2);
System.out.println("Point Distance: "+dist);

p2.move(5,-2);          // ==> (8,2)
p1.moveToXY(8,2);       // ==> (8,2)

if (p1.isEqualTo(p2))              // True!
    System.out.println("The two points are equal");
```

Output:

```
(0,0)
(3,4)
Point Distance: 5.0
The two points are equal
```

Some help:

- The class Point should of course be able to handle other points with different (x,y) values.
- The coordinates (x,y) are always integers and *can* be negative.
- The method `toString()` returns a string with coordinates suitable for print-outs.
- Distance between two points is computed in the same way as in Assignment 1.
- Two points are equal if they have the same coordinates.
- Method move moves the point certain steps in x- and y-direction.
- Method `moveToXY()` provide a new set of coordinates.

3.6 Date class

It is time to revisit the the task on date format from assignment two, but this time we will build a class called `DateFormat` instead of only having a method. The base requirements are the same as before with the addition that we want to make it possible to specify what punctuation to print between the numbers and it should be possible to choose anything – even nothing. As it is not possible to send an empty character, we use an exclamation mark (for "not") as no space between the numbers. The year must be sent with four digits and they can be assumed to be between 1900 and 2100. The following instantiation should be possible, where `!` is for no space between the numbers and `b` for big endian:

```
DateFormat date = new DateFormat(1977, 5, 25, '!', 'b');
```

An additional method needs to be created for returning the date as a string and it should have the following signature:

```
public String getDate(boolean fullYear)
```

In this call, you specify whether the year should be printed in full (like 1977) or just as the last to digits (77). The program below should work and give the output shown below it.

```
public class DateMain {
    public static void main(String[] args) {
        DateFormat date1 = new DateFormat(1977, 5, 25, '!', 'b');
        DateFormat date2 = new DateFormat();
        date2.setYear(1980);
        date2.setMonth(5);
        date2.setDay(17);
        date2.setPunctuation('/');
        date2.setFormat('l');
        DateFormat date3 = new DateFormat(1983, 5, 25, '-', 'm');
        DateFormat date4 = new DateFormat(1999, 13, 34, '-', 'm');

        System.out.println(date1.getDate(false));
        System.out.println(date2.getDate(true));
        System.out.println(date3.getDate(false));
        System.out.println(date4.getDate(true));
    }
}
```

Output:

```
770525
17/05/1980
05-25-83
Invalid date
```

Notice: be sure to encapsulate the attributes, both in the method and in the constructor. If an invalid date is sent to the object, set the part to zero and return “Invalid date” when printing, as seen above. However, you do not need to check for a valid month and day combination, that is next task and only for VG.

3.7 Valid dates – VG-task

Exercise 3.7 is marked as VG task → only mandatory for those of you that aspire for a higher mark (A or B).

Refine the date class to better check if a date is a valid combination of month with number of days. The first part of this is to check for months with 30 or 31 days, but do not forget to check for

leap years and February. There are a number of methods available in Java for checking if a year is a leap year – but you *may not* use any of those available in `LocalDate` or `GregorianCalendar` (other than for testing purposes). When the class is done, the test program below should run with the output show below it.

```
public class MoreDates {
    public static void main(String[] args) {
        DateFormat date = new DateFormat(2010, 4, 31, '/', 'b');
        DateFormat date1 = new DateFormat(2016, 2, 29, '/', 'b');
        DateFormat date2 = new DateFormat(2015, 2, 29, '/', 'm');
        DateFormat date3 = new DateFormat(2014, 2, 29, '/', 'l');
        DateFormat date4 = new DateFormat(2020, 2, 29, '/', 'l');
        DateFormat date5 = new DateFormat(1900, 2, 29, '-', 'b');

        System.out.println(date.getDate(false));
        System.out.println(date1.getDate(true));
        System.out.println(date2.getDate(false));
        System.out.println(date3.getDate(true));
        System.out.println(date4.getDate(false));
        System.out.println(date5.getDate(true));
    }
}
```

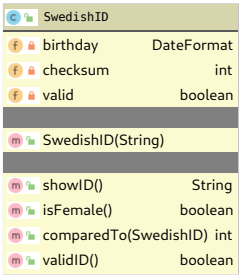
Output:

```
Invalid date
2016/02/29
Invalid date
Invalid date
29/02/20
Invalid date
```

3.8 Swedish ID – VG-task

Exercise 3.8 is marked as VG task → only mandatory for those of you that aspire for a higher mark (A or B).

In Sweden the most common way to identify yourself is by the *personnummer*, "personal number". This is a number that everyone gets if you are born in Sweden or move here more permanently. To the right is a class that you are going to implement in this task and the class represents a personal number (called



SwedishID). More information on the Swedish personal number can be seen on Wikipedia at https://en.wikipedia.org/wiki/Personal_identity_number_(Sweden). Valid personal numbers can be generated from the web page https://www.personnummer.nu/, which is in Swedish but it should not be too problematic to understand or translate it using Google Translate (or other similar service).

As you see, you need to reuse the refined class from the previous task in this for holding the birthday part of the ID. The complete number also has a *checksum* (shown as an attribute) which are four numbers where the last is a digit that says if the personal number is valid or not. See more information on this on the Wikipedia page. Also notice that you only have *one* constructor in this class, you will not be able to create empty objects.

Lastly, there are four methods which you are to implement:

- `public String showID()` – just returns the entire ID as a string on the format YYMMDD-XXXX.
- `public boolean isFemale()` – the third digit in the checksum specifies if the holder is male or female, see Wikipedia and implement this method that returns `true` if the ID is for a female.
- `public int comparedTo(SwedishID otherID)` – compares the ID with another ID. Notice that the name of the method is ***comparedTo()*** and not the usual `compareTo()`. The latter would be the correct way of doing it, but we will wait with *interfaces* till the next course. Other than that, it should work the same way as for strings, returning 1 (or positive) if this object is larger than the other, 0 if they are the same and -1 (or negative) otherwise.
- `public boolean validID()` – uses the Luhn algorithm to calculate if the checksum is correct. More on this is possible to read on the Wikipedia page.

Below is a program with a number of usages of the class and these need to work as described in the output. You need to provide a program called `SwedishIDMain.java` to show that your methods work, but it does not have to look exactly like the one below.

```java
public class SwedishIDMain {
    public static void main(String[] args) {
        SwedishID id1 = new SwedishID("19270414-4506");
        SwedishID id2 = new SwedishID("19270414-8671");
        SwedishID id3 = new SwedishID("19801214-7727");
        SwedishID id4 = new SwedishID("19811218-9876");
        SwedishID id5 = new SwedishID("19770525-1111");

        System.out.print(id1.showID() + " is a ");
        if(id1.isFemale())
            System.out.println("female.");
        else
            System.out.println("male.");

        System.out.print(id2.showID() + " is a ");
        if(id2.isFemale())
            System.out.println("female.");
        else
            System.out.println("male.");

        if(id1.comparedTo(id2) == 0)
            System.out.println("Yes, " + id1.showID() + " and " + id2.showID() + " are the
            ↪  same.");

        if(id1.comparedTo(id3) < 0)
            System.out.println(id1.showID() + " is older than " + id3.showID());

        if(id3.comparedTo(id1) > 0)
            System.out.println(id3.showID() + " is younger than " + id1.showID());

        if(id1.validID())
            System.out.println(id1.showID() + " is valid");
        else
            System.out.println(id1.showID() + "is not valid");

        if(id4.validID())
            System.out.println(id4.showID() + " is valid");
        else
            System.out.println(id4.showID() + " is not valid");

        if(id5.validID())
            System.out.println(id5.showID() + " is valid");
        else
            System.out.println(id5.showID() + " is not valid");
    }
}
```

Output:

```
19270414-4506 is a female.
19270414-8671 is a male.
Yes, 19270414-4506 and 19270414-8671 are the same.
19270414-4506 is older than 19801214-7727
19801214-7727 is younger than 19270414-4506
19270414-4506 is valid
19811218-9876 is valid
19770525-1111 is not valid
```

3.9 Game mechanics

The last task is to construct the basis of a simple game mechanic. The mechanic will involve two opponents fighting each others – if you feel at all *uncomfortable* with this, you may change the parts you do not like. You could, for instance, make this about cars avoiding obstacles or horses trying to jump over fences. The important part is that the behaviour is the same.

We are also not giving you a class diagram or any of the classes as such. In this task you must define what classes with what properties and behaviours are useful to solve the task. To help you, we will provide you with a description of what the game is about as well as a print out of a possible test run of the program. You will probably need to make assumptions, and that is okay. The important part is that you think in an *object oriented way* in your solution.

> *In this game there will be heroes and enemies fighting each others in a fantasy environment. The fighting mechanics is loosely based on old time role playing games where ordinary six sided dice were used to decided the outcome of an encounter. Both heroes and enemies have a set of dice for attacking and defending themselves. The difference is that the hero uses six sided dice while we want the enemy to only have five sided dice (most of the time we want the hero to win). In a fight the attack of one part is set against the defence of the other part. The number of dice for each property must be set at instantiation.*
>
> *An example, an hero has three dice for attack and with them he hits 9 (for example 3 + 5 + 1). The enemy has four dice for defence and with them he hits 8 (for example 1 + 4 + 2 + 1). This means that the attack from the hero goes through and the hero can roll damage. Damage is done in the same way, namely with a number of dice. The number of dice is decided by the weapon that the hero has. There can be any number of weapons in the game, but they are all having the same basic structure; a name and a power which is the number of six sided dice to*

roll for the damage. As the hero in our example hit the enemy, the damage from the weapon is calculated, for example if the power of the weapon is three, three six sided dice are rolled – say to 8 (1 + 6 + 1). This amount is reduced from the XP (experience points) that the enemy has. If the number of XP is below zero, the enemy is defeated.

When it comes to weapons, only heroes can have them, with the emphasis on can, as they can be without weapons. An hero can pick up a weapon and then, and only then, does he/she have the weapon. If the hero does not have a weapon, the damage should be a random number between two and four. The enemies do not carry any weapons at all and therefore only has a simple random number between 1 and 14 for damage.

Sketch on the classes needed (you don't need to hand your sketches in) and what each of them need to be useful (and provide the behaviour necessary). The classes need constructors and it is quite okay to only create constructors with parameters. Notice that you *may not* use inheritance in this task as we have not covered that yet.

A part from the different classes you also need to provide a class with a main method called `GameManager.java` in which you prove that the classes work by creating one hero and one enemy and at least one weapon. The opponents should fight each others in a number of rounds (where each round is one attack from the hero, defence from the enemy, one attack from the enemy and defence from the hero). You may include more, but you must show at least one fight till one of the opponents is defeated. You can look at the following example run of a test program to know more or less what you need to provide. In the example, Lord Gurka is the hero and Prince Purjo is the enemy. The hero has a weapon, a Steel Sallad, that he uses to fight the prince. By commenting out that Lord Gurka is using the weapon, he is instead using his fists.

```
Lord Gurkan has 50 XP and a Steel Sallad to fight with.
Prince Purjo has 30 XP and uses his fists to fight with.

Round 1:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 2:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 3:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 4:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 5:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 6:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 7:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 8:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 9:
Lord Gurkan gives Prince Purjo a powerful blow!
Prince Purjo is still alive!
```

```
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 10:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo manages to hit Lord Gurkan!
Lord Gurkan is still well and kicking!

Round 11:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 12:
Lord Gurkan gives Prince Purjo a powerful blow!
Prince Purjo is still alive!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 13:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 14:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo manages to hit Lord Gurkan!
Lord Gurkan is still well and kicking!

Round 15:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo manages to hit Lord Gurkan!
Lord Gurkan is still well and kicking!

Round 16:
Lord Gurkan gives his all, but Prince Purjo dodges!
Prince Purjo tries his best but Lord Gurkan dodges the blow!

Round 17:
Lord Gurkan gives Prince Purjo a powerful blow!
Prince Purjo is defeated!
```