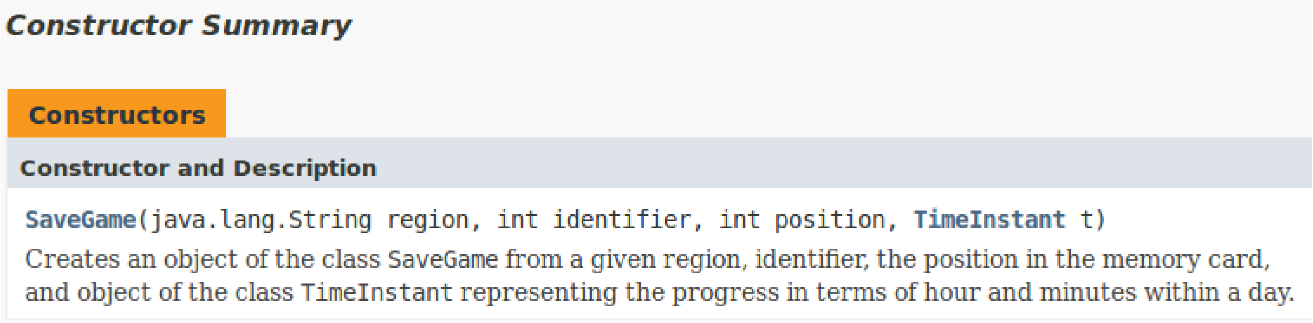
Second mid term exam – IIP (ETSInf)
January 7th, 2019 – Duration: 2 hours and 30 minutes
Notice: This exam is evaluated over 10 points, but its weight in the final grade of IIP is **3,6 points**

NAME:	GROUP:

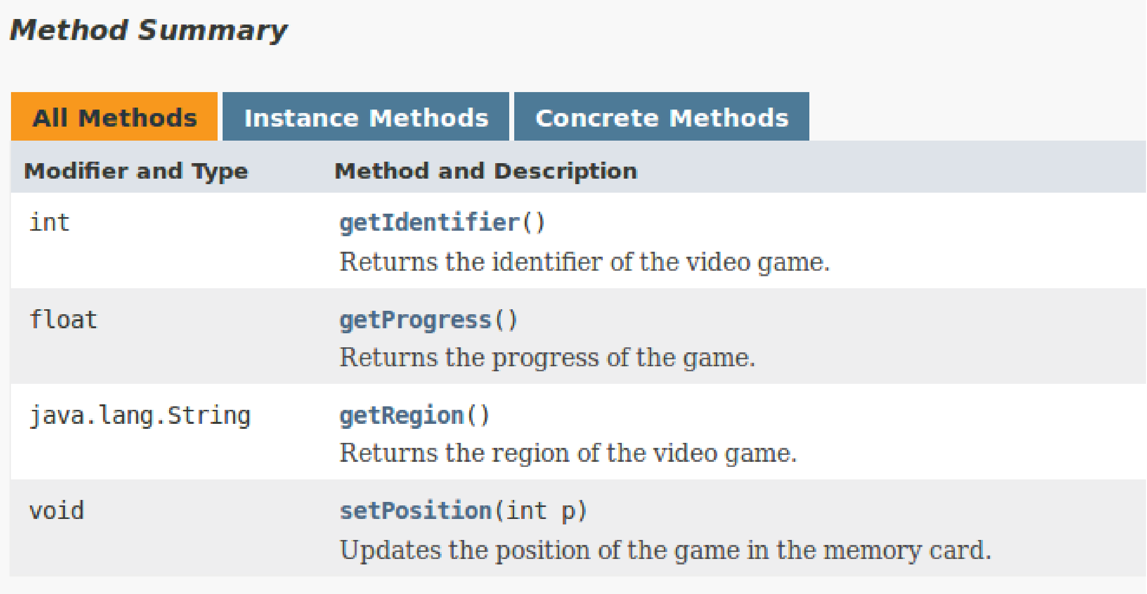
1. 6 points The class `SaveGame` is already implemented and available. Objects of this class represent a game of the video game stored in a memory card of a well-known video-game console. Each object of the class `SaveGame` has four attributes: the region code of the video game, the identifier of the video game within its region, the position in the memory card, and the percentage of the achieved progress. This class has been used in other examples and exercises and here you have part of its documentation:



Constructor Summary

Constructors

Constructor and Description
`SaveGame(java.lang.String region, int identifier, int position, TimeInstant t)` Creates an object of the class SaveGame from a given region, identifier, the position in the memory card, and object of the class `TimeInstant` representing the progress in terms of hour and minutes within a day.



Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`int`	`getIdentifier()` Returns the identifier of the video game.
`float`	`getProgress()` Returns the progress of the game.
`java.lang.String`	`getRegion()` Returns the region of the video game.
`void`	`setPosition(int p)` Updates the position of the game in the memory card.

You have to implement the class `SaveArea` (a data-type class) for storing the games within a video-game console. This class will be part of the video-game console software and will have the following attributes and methods:

a) (0,5 points) Attributes:

- `MAX_STORED`: a class and constant attribute of type `int` set to 100, the maximum number of games that can be stored in the memory card of the video-game console.
- `size`: private instance variable or attribute indicating the number of games stored in the memory card.
- `storedGames`: private instance variable or attribute for storing the objects of the class `SaveGame`. This attribute must be an array with a capacity equal to `MAX_STORED`. The stored games must be located in consecutive positions from 0 to `size-1`. Remember that the attribute position of objects of the class `SaveGame` contain the position of the stored game in this array, so this detail must be taken into account when implementing methods `removeTheOldestOne()` and `save()` because the attribute position of the objects of the class `SaveGame` must be updated properly.

b) (0,5 points) A default constructor without parameters for creating the array and set the size equal to zero because, initially, there are no games stored in the memory card.

c) (1 point) A method with the following profile:

```
private void removeTheOldestOne()
```

that removes the oldest stored game by shifting one position to the left all the stored games which are to the right of it. This method does nothing if the array is empty.

d) (1 point) A method with the following profile:

```
private boolean withAProgressGreaterThanOrEqualTo( SaveGame sg )
```

that returns `true` if already exists a game with the same identifier and with a progress greater than or equal to the progress of `sg`, and returns `false` otherwise.

e) (1,5 points) A method with the following profile:

```
public boolean save( SaveGame sg )
```

that adds `sg` to the previous stored games. To do it the following conditions must hold:

1 its progress must be greater than the progress of other games previously stored with the same identifier. This method can check it by using the method `withAProgressGreaterThanOrEqualTo()`.

2 there are free positions in the array. If the array is full, before adding the new game the oldest game must be removed by using the method `removeTheOldestOne()`, even if the oldest game corresponds to a distinct video game (with different identifier).

This method returns `true` when the new game has been stored successfully, otherwise returns `false`.

f) (1,5 points) A method with the following profile:

```
public SaveGame [] filterByRegion( String region )
```

that returns an array of objects of the class `SaveGame` with those games whose video games belong to the indicated region via the parameter `region`. If there are no games corresponding to that region in the array, then an empty array must be returned.

Solution:

```
public class SaveArea
{
    public static final int MAX_STORED = 100;

    private SaveGame [] storedGames;
    private int         size;

    public SaveArea()
    {
        storedGames = new SaveGame[ MAX_STORED ];
        size = 0;
    }

    private void removeTheOldestOne()
    {
        if ( size > 0 ) {
            for( int j = 0; j < size - 1; j++ ) {
                storedGames[j] = storedGames[j + 1];
                storedGames[j].setPosition(j);
            }
            storedGames[--size] = null;
        }
    }

    private boolean withAProgressGreaterThanOrEqualTo( SaveGame sg )
    {
        boolean sameId = false;

        int i = size - 1;

        while( i >= 0 && !sameId ) {

            if ( sg.getIdentifier() == storedGames[i].getIdentifier() ) {
                sameId = true;
            } else {
                i--;
            }
        }

        return sameId  &&  storedGames[i].getProgress() >= sg.getProgress();
    }
    public boolean save( SaveGame sg )
    {
        // 'sg' is not stored if already exists another game of the same video game
        // with a progress greater than or equal to the progress of 'sg'
        if ( withAProgressGreaterThanOrEqualTo(sg) ) {
            return false;
        }

        // Removes the oldest game if necessary
        if ( size == storedGames.length ) {
            removeTheOldestOne();
        }

        // Stores the new game
        storedGames[size] = sg;
        sg.setPosition(size);
        ++size;

        return true;
    }



    public SaveGame [] filterByRegion( String region )
    {
        int counter = 0;

        for( int i = 0; i < size; i++ ) {

            if ( storedGames[i].getRegion().equals(region) ) {
                ++counter;
            }
        }

        SaveGame [] res = new SaveGame[counter];

        int j = 0;
        for( int i = 0; i < size && j < res.length; i++ ) {

            if ( storedGames[i].getRegion().equals(region) ) {
                res[j++] = storedGames[i];
            }
        }
        return res;
    }

    public void show()
    {
        System.out.println();
        System.out.println( "--------------------------------------------" );
        for( int i = 0; i < size; i++ ) System.out.println( storedGames[i] );
        System.out.println( "--------------------------------------------" );
        System.out.println();
    }
}
```

2. 2 points A polygonal number is a natural number that can be distributed in a regular polygon with s sides. For instance, nine is a square number and six is a triangular number:

```
o o o        o
o o o       o o
o o o      o o o
```

In general, polygonal numbers follow the formula:

$$n \cdot \frac{(s-2) \cdot n - (s-4)}{2}$$

where n is the n-th polygonal number and s is the number of sides of the polygon. For instance, for $s = 3$, the polygonal numbers are: $1, 3, 6, 10, 15, 21, 28, \ldots$

You have to write an static method that given a natural number $k > 0$ and the number of sides of the polygon s, returns `true` if k is a s-polygonal number, otherwise returns `false`. For instance, if $k = 15$ and $s = 3$, the methods returns `true` because `15` is the $5th$ three-polygonal number, but if $k = 19$ and $s = 3$, the method returns `false` because `19` is not a three-polygonal number.

Solution:

```
public static boolean isAPolygonalNumber( int k, int sides )
{
    int n = 1;
    int i = 2;

    while( n < k ) {
        n = i * ((sides-2)*i - (sides-4)) / 2;
        i++;
    }
    return n == k;
}
```

3. 2 points **You have to** write an static method with two arrays as parameters: `edges`, an `int` array, and `values`, a `double` array. The result must be an array of type `int` with the same length than `edges`, where the element at position i must be the counter of values in the array `values` lower than the value stored at position i in the array `edges`.

For instance, if the parameters are the following ones

- `edges = `{`15, 35, 50, 37, 25, 70`}
- `values = `{`10.0, 20.0, 50.0, 40.0, 30.0, 80.0`}

the resulting array should be {`1, 3, 4, 3, 2, 5`}, because there are

- `1` value less than `15` (`10.0`)
- `3` values less than `35` (`10.0`, `20.0`, `30.0`)
- `4` values less than `50` (`10.0`, `20.0`, `30.0`, `40.0`)
- `3` values less than `37` (`10.0`, `20.0`, `30.0`)
- `2` values less than `25` (`10.0`, `20.0`)
- `5` values less than `70` (`10.0`, `20.0`, `30.0`, `40.0`, `50.0`)

Solution:

```
public static int[] accumulatedFrequency( int [] edges, double [] values )
{
    int [] res = new int [ edges.length ];

    for( int i = 0; i < values.length; i++ ) {
        for( int j = 0; j < edges.length; j++ ) {
            if ( values[i] < edges[j] ) {
                res[j]++;
            }
        }
    }

    return res;
}
```