# Retake exam for IIP Second Partial - ETSInf.
# Date: January, 23rd, 2015. Time: 2:30 hours.

1. 6.5 points A shipping company must manage their shipping distributions, that could be packages or documents. Each shipping good has as data its origin (`String`), its destination (`String`), a reference (`String`) that acts as an identifier, a type (`int`) and a situation (`String`). A shipping good can be stored in a storehouse, where a specific situation or shelf is assigned to the good while the good is in the storehouse. When the shipping good is taken from the storehouse, it is unregistered by changing its situation to `"In transit"`.

   For representing the problem, a class `ShippingGood` is available. Below an extract of its documentation is shown:

### Field Summary

**Fields**

| Modifier and Type | Field and Description |
|---|---|
| `static int` | **`DOCUMENT`**<br>Constant that indicates that the good is a document. |
| `static int` | **`PACKAGE`**<br>Constant that indicates that the good is a package. |

### Constructor Summary

**Constructors**

| Constructor and Description |
|---|
| **`ShippingGood`**`(java.lang.String origin, java.lang.String destination, java.lang.String ref, int type)`<br>Creates a good with given origin, destination, reference and type (PACKAGE or DOCUMENT), and with default situation "In transit". |

### Method Summary

**Methods**

| Modifier and Type | Method and Description |
|---|---|
| `boolean` | **`equals`**`(java.lang.Object o)`<br>Checks if a good is equal to the other, that is, their references are the same. |
| `java.lang.String` | **`getDestination`**`()`<br>Returns destination of the good. |
| `java.lang.String` | **`getOrigin`**`()`<br>Returns origin of the good. |
| `java.lang.String` | **`getRef`**`()`<br>Returns reference of the good. |
| `java.lang.String` | **`getSituation`**`()`<br>Returns situation of the good. |
| `int` | **`getType`**`()`<br>Returns type of the good. |
| `void` | **`setSituation`**`(java.lang.String newSit)`<br>Updates to newSit the shipping good situation. |
| `java.lang.String` | **`toString`**`()`<br>Returns String with shipping good data. |

   You can see that a `ShippingGood` object is created from the `String` parameters that indicate their `origin`, `destination`, and `reference`, and the `int` parameter that indicates its `type`. As said above, a `ShippingGood` has another attribute `String situation`, that is inited to `"In transit"` when it is created. The class has two `int` public constants, `DOCUMENT` and `PACKAGE`, inited to the values that correspond to the proper types of shipping goods. Notice that consultor (`get`) methods are available for all data of `ShippingGood`, and that there is a modifier (`set`) method for its situation.

   **You must:** implement the class `StoreHouse` that represents a storehouse with shipping goods by using the following components (attributes and methods):

   a) (0.5 points) Attributes:

   - `MAX_GOODS`, class (`static`) constant that represents the maximum number of goods that can be stored in the storehouse; its value must be 2000; remember that this constant, along with the `ShippingGood` constants, must be used whenever is needed

- `numG`, integer number in the interval `[0..MAX_GOODS]` that represents the number of stored goods in the storehouse in a given moment
- `list`, array of `ShippingGood`, of size `MAX_GOODS`; the elements of this array are the goods in the storehouse, and they are stored in positions from `0` to `numG-1`, and by arrival (i.e., `list[0]` is the oldest good from all that are currently stored in the array, and `list[numG-1]` is the most recent); there are no repeated goods
- `numDoc` and `numPac`, integers that represent how many goods at a given moment are of type document or package, respectively

Methods that store into or take from the storehouse the different goods must keep the goods consecutive in the array, by arrival order, and must update properly all the counters.

b) (0.5 points) A default constructor (without parameters) that creates an empty storehouse (with 0 goods)

c) (1 point) A method with header:

```
private int positionOf(String ref)
```

that returns the position of the array `list` where the good with reference `ref` appears, or -1 when it is not present. This is an utility method that must be used in the methods `store`, `search`, and `takeFrom` that are asked below

d) (1 point) A method with header:

```
public boolean store(ShippingGood g, String shelf)
```

that updates the storehouse by storing the good `g`. The good `g` itself must store the information on its situation; thus, the situation of `g` must be modified to that given by the parameter `String shelf`. The good will not be stored if there is any other that it is equal to it (same reference) or if the storehouse is full. When the good is not stored, `false` is returned, and `true` otherwise.

e) (0.75 points) A method with header:

```
public ShippingGood search(String ref)
```

that returns the good in the storehouse whose reference is `ref`. If it is not found, it will return `null`.

f) (1 point) A method with header:

```
public ShippingGood takeFrom(String ref)
```

that searches the `ShippingGood` with reference `ref`; when it is found, it changes its situation to `"In transit"`, it takes it from the storehouse, and returns it as a result; when no good for that reference is found, it returns `null`.

Remember that the method must keep the goods as consecutive elements in the first positions of the array `list`, and keeping the arrival oder criterium, thus that when taking out a good from the array, all the elements at the right of it will be moved one position to the left

g) (1 point) A method with header:

```
public ShippingGood[] obtainGoods(int type)
```

that returns an array with the `ShippingGood` that are of the type indicated by the parameter `type`. The length of this array must be equal to the number of goods of that type, or 0 if no goods of that type are present or the type is not equal to any of the valid types defined in `ShippingGood`

h) (0.75 points) A method with header:

```
public String toString()
```

that returns a `String` with the data of all the goods that are in the storehouse, by arrival order, each one in a different line

**Solution:**

```
public class StoreHouse {
    public static final int MAX_GOODS = 2000;
    private ShippingGood[] list;
    private int numG, numDoc, numPac;

    public StoreHouse() {
      list = new ShippingGood[MAX_GOODS];
      numG = numDoc = numPac = 0;
    }

    private int positionOf(String ref) {
        int i = 0;
        while (i < numG && !ref.equals(list[i].getRef())) i++;
        if (i < numG) return i; else return -1;
    }
```

```
    public boolean store(ShippingGood g, String shelf) {
        if (positionOf(g.getRef()) == -1 || numG == MAX_GOODS) return false;
        g.setSituation(shelf);
        list[numG++] = g;
        if (g.getType() == ShippingGood.PACKAGE) numPac++; else numDoc++;
        return true;
    }

    public ShippingGood search(String ref) {
        int pos = positionOf(ref);
        if (pos != -1) return list[pos]; else return null;
    }

    public ShippingGood takeFrom(String ref) {
        int pos = positionOf(ref);
        if (pos == -1) return null;
        ShippingGood g = list[pos];
        g.setSituation("In transit");
        for (int i = pos + 1; i < numG; i++) list[i - 1] = list[i];
        numG--;
        if (g.getType() == ShippingGood.PACKAGE) numPac--; else numDoc--;
        return g;
    }

    public ShippingGood[] obtainGoods(int type) {
        int size = 0;
        switch (type) {
            case ShippingGood.DOCUMENT: size = numDoc; break;
            case ShippingGood.PACKAGE: size = numPac; break;
        }
        ShippingGood[] lGo = new ShippingGood[size];
        for (int i = 0, j = 0; j < size; i++)
            if (list[i].getType() == type) {
                lGo[j] = list[i];
                j++;
            }
        return lGo;
    }

    public String toString(){
        String result = "List of the goods in the storehouse:\n";
        for (int i = 0; i < numG; i++)
            result += list[i] + "\n";
        return result;
    }
}
```

2. 1.75 points **You must:** Implement a class (`static`) method that, given an array `a` of `double` (`a.length` $> 0$), and an integer `n` (`n` $> 0$), modifies all the elements in the array in such a way tha the maximum value in the array is `n` and the rest of the elements get scaled according to this vale. That is, if the maximum in `a` is `m`, the relation between the new and the old value of each element in the array must be `n/m`. For example, if `a` is {`4.5, 27.0, 18.0, 1.5`} and `n = 9`, `a` must change to {`1.5, 9.0, 6.0, 0.5`}.

**Solution:**

```
/** a.length > 0, a[i] > 0 ∀i:  0 <= i < a.length, n > 0 */
public static void escale(double[] a, int n) {
    double max = a[0];
    for (int i = 1; i < a.length ; i++)
        if (a[i] > max) max = a[i];
    for (int j = 0; j < a.length; j++)
        a[j] = (a[j] / max) * n;
}
```

3. [1.75 points] Bertrand's postulate is: "If $n$ is a natural number greater than 3, it always exists a prime number $p$ such that $n < p < 2n - 2$". In a class you have available a method with header:

```
/** n > 1 */
public static boolean is_prime(int n)
```

that given `n` greater than 1, checks if it is a prime number.

**You must:** Implement a class (`static`) method (that would be in the same class) that, given a natural number `n` with `n`>3, determines by using the `is_prime` method, which prime number `p` fulfills the Bertrand's postulate for `n`. In case there is more than one, it must return the lowest one. For example, for `n=8`, the primes in the interval `[9,13]` are 11 and 13, so the method must return 11.

**Solution:**

```
/** n > 3 */
public static int bertrand(int n) {
    int p = n + 1;
    boolean found = false;
    while (p < (2 * n - 2) && !found)
           if (is_prime(p)) found = true;
       else p++;
    return p;
}
```