Programming exercise:

ABC (2 parts)

Let's practice creating and inheriting classes.

Creating classes

Create the following three classes:

- Class `A`. Class should have no object variables nor should you specify a constructor for it. It only has the method `public void a()`, which prints a string "A".
- Class `B`. Class should have no object variables nor should you specify a constructor for it. It only has the method `public void b()`, which prints a string "B".
- Class `C`. Class should have no object variables nor should you specify a constructor for it. It only has the method `public void c()`, which prints a string "C".

```
A a = new A();
B b = new B();
C c = new C();

a.a();
b.b();
c.c();
```

Sample output

```
A
B
C
```

Class inheritance

Modify the classes so that class B inherits class A, and class C inherits class B. In other words, class A will be a superclass for class B, and class B will be a superclass for class C.

```
C c = new C();

c.a();
c.b();
c.c();
```

Sample output

```
A
B
C
```

Programming exercise:

Person and subclasses (5 parts)

Person

Create a class `Person`. The class must work as follows:

```
Person ada = new Person("Ada Lovelace", "24 Maddox St. London W1S 2QN");
Person esko = new Person("Esko Ukkonen", "Mannerheimintie 15 00100 Helsinki");
System.out.println(ada);
System.out.println(esko);
```

Sample output

```
Ada Lovelace
  24 Maddox St. London W1S 2QN
Esko Ukkonen
  Mannerheimintie 15 00100 Helsinki
```

Student

Create a class `Student`, which inherits the class `Person`.

At creation, a student has 0 study credits. Every time a student studies, the amount of study credits goes up. The class must act as follows:

```
Student ollie = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 90028");
System.out.println(ollie);
System.out.println("Study credits " + ollie.credits());
ollie.study();
System.out.println("Study credits "+ ollie.credits());
```

Sample output

```
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
Study credits 0
Study credits 1
```

Student's toString

In the previous task, `Student` inherits the toString method from the class `Person`. However, you can also overwrite an inherited method, replacing it with your own version. Write a version of toString method specifically for the `Student` class. The method must act as follows:

```
Student ollie = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 90028");
System.out.println(ollie);
ollie.study();
System.out.println(ollie);
```

Sample output

```
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  Study credits 0
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  Study credits 1
```

Teacher

Create a class `Teacher`, which inherits the class `Person`.

The class must act as follows:

```
Teacher ada = new Teacher("Ada Lovelace", "24 Maddox St. London W1S 2QN", 1200);
Teacher esko = new Teacher("Esko Ukkonen", "Mannerheimintie 15 00100 Helsinki", 5400
System.out.println(ada);
System.out.println(esko);

Student ollie = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 90028");

int i = 0;
while (i < 25) {
  ollie.study();
  i = i + 1;
}
System.out.println(ollie);
```


Sample output

```
Ada Lovelace
  24 Maddox St. London W1S 2QN
  salary 1200 euro/month
Esko Ukkonen
  Mannerheimintie 15 00100 Helsinki
  salary 5400 euro/month
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  Study credits 25
```

List all Persons

Write a method `public static void printPersons(ArrayList<Person> persons)` in the Main class. The method prints all the persons on the list given as the parameter. Method must act as follows when invoked from the `main` method:

```
public static void main(String[] args) {
    ArrayList<Person> persons = new ArrayList<Person>();
    persons.add(new Teacher("Ada Lovelace", "24 Maddox St. London W1S 2QN", 1200));
    persons.add(new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 90028"));

    printPersons(persons);
}
```

Sample output

```
Ada Lovelace
  24 Maddox St. London W1S 2QN
  salary 1200 euro/month
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  Study credits 0
```

Programming exercise:
Warehousing (7 parts)

The exercise template contains a class `Warehouse`, which has the following constructors and methods:

- **public Warehouse(double capacity)** - Creates an empty warehouse, which has the capacity provided as a parameter; an invalid capacity (<=0) creates a useless warehouse, with the the capacity 0.
- **public double getBalance()** - Returns the balance of the warehouse, i.e. the capacity which is taken up by the items in the warehouse.
- **public double getCapacity()** - Returns the total capacity of the warehouse (i.e. the one that was provided in the constructor).
- **public double howMuchSpaceLeft()** - Returns a value telling how much space is left in the warehouse.
- **public void addToWarehouse(double amount)** - Adds the desired amount to the warehouse; if the amount is negative, nothing changes, and if everything doesn't fit, then the warehouse is filled up and the rest is "thrown away" / "overflows".
- **public double takeFromWarehouse(double amount)** - Take the desired amount from the warehouse. The method returns much we actually **get**. If the desired amount is negative, nothing changes and we return 0. If the desired amount is greater than the amount the warehouse contains, we get all there is to take and the warehouse is emptied.
- **public String toString()** - Returns the state of the object represented as a string like this `balance = 64.5, space left 123.5`

In this exercise we build variations of a warehouse based on the `Warehouse` class.

Product warehouse, step 1

The class `Warehouse` handles the functions related to the amount of a product. Now we want a product name for the product and a way to handle the name. **Let's write ProductWarehouse as a subclass of Warehouse!** First, we'll just create a private object variable for the product name, a constructor, and a getter for the name field:

- **public ProductWarehouse(String productName, double capacity)** - Creates an empty product warehouse. The name of the product and the capacity of the warehouse are provided as parameters.
- **public String getName()** - Returns the name of the product.

Remind yourself of how a constructor can run the constructor of the superclass as its first action!

Example usage:

```
ProductWarehouse juice = new ProductWarehouse("Juice", 1000.0);
juice.addToWarehouse(1000.0);
juice.takeFromWarehouse(11.3);
System.out.println(juice.getName()); // Juice
System.out.println(juice);           // balance = 988.7, space left 11.3
```

Sample output

```
Juice
balance = 988.7, space left 11.3
```

Product warehouse, step 2

As we can see from the previous example, the `toString()` inherited by the ProductWarehouse object doesn't (obviously!) know anything about the product name. *Something must be done!* Let's also add a setter for the product name while we're at it:

- **public void setName(String newName)** - sets a new name for the product.
- **public String toString()** - Returns the state of the object represented as a string like this `Juice: balance = 64.5, space left 123.5`

The new `toString()` method could be written using the getters inherited from the superclass, which would give access to values of inherited, but still hidden fields. However, the superclass already has the desired functionality to provide a string representation of the warehouse state, so why bother recreating that functionality? Just take advantage of the inherited `toString()`.

Remind yourself of how to call an overridden method in a subclass!

Usage example:

```
ProductWarehouse juice = new ProductWarehouse("Juice", 1000.0);
juice.addToWarehouse(1000.0);
juice.takeFromWarehouse(11.3);
System.out.println(juice.getName()); // Juice
juice.addToWarehouse(1.0);
System.out.println(juice);           // Juice: balance = 989.7, space left 10.29999
```

Sample output

```
Juice
Juice: balance = 989.7, space left 10.299999999999955
```

Change History, step 1

Sometimes it might be useful to know how the inventory of a product changes over time: Is the inventory often low? Are we usually at the limit? Are the changes in inventory big or small? Etc. Thus we should give the `ProductWarehouse` class the ability to remember the changes in the amount of a product.

Let's begin by creating a tool that aids in the desired functionality.

The storing of the change history could of course have been done using an `ArrayList<Double>` object in the class *ProductWarehouse*, however, we want our own *specialized tool* for this purpose. The tool should be implemented by encapsulating the `ArrayList<Double>`object.

Public constructors and methods of the `ChangeHistory` class:

- **public ChangeHistory()** creates an empty `ChangeHistory` object.
- **public void add(double status)** adds provided status as the latest amount to remember in the change history.
- **public void clear()** empties the history.
- **public String toString()** returns the string representation of the change history. *The string representation provided by the ArrayList class is sufficient.*

Change History, step 2

Build on the `ChangeHistory` class by adding analysis methods:

- **public double maxValue()** returns the largest value in the change history. If the history is empty, the method should return zero.
- **public double minValue()** returns the smallest value in the change history. If the history is empty, the method should return zero.
- **public double average()** returns the average of the values in the change history. If the history is empty, the method should return zero.

The methods should not modify the order of the encapsulated list.

Product warehouse with history, step 1

Implement `ProductWarehouseWithHistory` as a subclass of `ProductWarehouse`. In addition to all the previous features this new warehouse also provides services related to the change history of the warehouse inventory. The history is managed using the `ChangeHistory` object.

Public constructors and methods:

- **public ProductWarehouseWithHistory(String productName, double capacity, double initialBalance)** creates a product warehouse. The product name, capacity, and initial balance are provided as parameters. Set the initial balance as the initial balance of the warehouse, as well as the first value of the change history.
- **public String history()** returns the product history like this `[0.0, 119.2, 21.2]`. Use the string representation of the ChangeHistory object as is.

NB in this initial version the history is not yet working properly; currently it only remembers the initial balance.

Usage example:

```
// the usual:
ProductWarehouseWithHistory juice = new ProductWarehouseWithHistory("Juice", 1000.0,
juice.takeFromWarehouse(11.3);
System.out.println(juice.getName()); // Juice
juice.addToWarehouse(1.0);
System.out.println(juice);           // Juice: balance = 989.7, space left 10.3

// etc

// however, history() still doesn't work properly:
System.out.println(juice.history()); // [1000.0]
// so we only get the initial state of the history set by the constructor...
```

Sample output

```
Juice
Juice: balance = 989.7, space left 10.299999999999955
[1000.0]
```

Product warehouse with history, step 2

It's time to make history! The first version didn't know anything but the initial state of the history. Expand the class with the following methods

- **public void addToWarehouse(double amount)** works just like the method in the `Warehouse` class, but we also record the changed state to the history. **NB:** the value recorded in the history should be the warehouse's balance after adding, not the amount added!
- **public double takeFromWarehouse(double amount)** works just like the method in the `Warehouse` class, but we also record the changed state to the history. **NB:** the value recorded in the history should be the warehouse's balance after removing, not the amount removed!

Usage example:

```
// the usual:
ProductWarehouseWithHistory juice = new ProductWarehouseWithHistory("Juice", 1000.0,
juice.takeFromWarehouse(11.3);
System.out.println(juice.getName()); // Juice
juice.addToWarehouse(1.0);
System.out.println(juice);           // Juice: balance = 989.7, space left 10.3

// etc

// and now we have the history:
System.out.println(juice.history()); // [1000.0, 988.7, 989.7]
```

Sample output

```
Juice
Juice: balance = 989.7, space left 10.299999999999955
[1000.0, 988.7, 989.7]
```

Remember how an overriding method can take advantage of the overridden method!

Product warehouse with history, step 3

Expand the class with the method

- **public void printAnalysis()**, which prints history related information for the product in the way presented in the example.

Usage example:

```
ProductWarehouseWithHistory juice = new ProductWarehouseWithHistory("Juice", 1000.0,
juice.takeFromWarehouse(11.3);
juice.addToWarehouse(1.0);
//System.out.println(juice.history()); // [1000.0, 988.7, 989.7]

juice.printAnalysis();
```

Sample output

```
Product: Juice
History: [1000.0, 988.7, 989.7]
Largest amount of product: 1000.0
Smallest amount of product: 988.7
Average: 992.8
```