Programming exercise:

DifferentKindsOfBoxes (3 parts)

In the exercise template you'll find the classes `Item` and `Box`. `Box` is an abstract class, where adding multiple items is implemented by repeatedly calling the `add`-method. The `add`-method, meant for adding a single item, is abstract, so every class that inherits it, must implement it. Your assignment is to edit the `Box`-class and to implement different kinds of boxes based on the `Box` class.

```java
import java.util.ArrayList;

public abstract class Box {

    public abstract void add(Item item);

    public void add(ArrayList<Item> items) {
        for (Item item : items) {
            Box.this.add(item);
        }
    }

    public abstract boolean isInBox(Item item);
}
```

Editing the Item class

Implement the `equals` and `hashCode` methods for the `Item`-class. They are needed, so that you can use the `contains`-methods of different lists and collections. Implement the methods in such a way that value of the `weight` instance variable of the `Item`-class isn't considered. *It's probably a good idea to make use of Netbeans's functionality to implement the `equals` and `hashCode` methods*

Box with a max weight

Implement the class `BoxWithMaxWeight`, that inherits the `Box` class. BoxWithMaxWeight has a constructor `public BoxWithMaxWeight(int capacity)`, that defines the max weight allowed for that box. You can add an item to a BoxWithMaxWeight when and only when, adding the item won't cause the boxes maximum weight capacity to be exceeded.

```
BoxWithMaxWeight coffeeBox = new BoxWithMaxWeight(10);
coffeeBox.add(new Item("Saludo", 5));
coffeeBox.add(new Item("Pirkka", 5));
coffeeBox.add(new Item("Kopi Luwak", 5));

System.out.println(coffeeBox.isInBox(new Item("Saludo")));
System.out.println(coffeeBox.isInBox(new Item("Pirkka")));
System.out.println(coffeeBox.isInBox(new Item("Kopi Luwak")));
```

Sample output

```
true
true
false
```

One item box and the misplacing box

Next, implement the class `OneItemBox`, that inherits the `Box` class. `OneItemBox` has the constructor `public OneItemBox()`, and it has the capacity of exactly one item. If there is already an item in the box, it must not be switched. The weight of the item added to the box is irrelevant.

```
OneItemBox box = new OneItemBox();
box.add(new Item("Saludo", 5));
box.add(new Item("Pirkka", 5));

System.out.println(box.isInBox(new Item("Saludo")));
System.out.println(box.isInBox(new Item("Pirkka")));
```

Sample output

```
true
false
```

Next implement the class `MisplacingBox`, that inherits the `Box`-class. MisplacingBox has a constructor `public MisplacingBox()`. You can add any items to a misplacing box, but items can never be found when looked for. In other words adding to the box must always succeed, but calling the method `isInBox` must always return false.

```java
MisplacingBox box = new MisplacingBox();
box.add(new Item("Saludo", 5));
box.add(new Item("Pirkka", 5));

System.out.println(box.isInBox(new Item("Saludo")));
System.out.println(box.isInBox(new Item("Pirkka")));
```

Sample output

```
false
false
```

Programming exercise:

TacoBoxes (2 parts)

In the exercise template you'll find Interface `TacoBox` ready for your use. It has the following methods:

- the method `int tacosRemaining()` return the number of tacos remaining in the box.
- the method `void eat()` reduces the number of tacos remaining by one. The number of tacos remaining can't become negative.

```
public interface TacoBox {
    int tacosRemaining();
    void eat();
}
```

Triple taco box

Implement the class `TripleTacoBox`, that implements the `TacoBox`interface. `TripleTacobox` has a constructor with no parameters. `TripleTacobox` has an object variable `tacos` which is initialized at 3 when the constructor is called.

Custom taco box

Implement the class `CustomTacoBox`, that implements the `TacoBox`interface. `CustomTacoBox` has a constructor with one parameter defining the initial number of tacos in the box(`int tacos`).

Programming exercise:

Interface In A Box (4 parts)

Packables

Moving houses requires packing all your belongings into boxes. Let's imitate that with a program. The program will have boxes, and items to pack into those boxes. All items must implement the following Interface:

```java
public interface Packable {
    double weight();
}
```

Add the Interface to your program. Adding a new Interface is quite similar to adding a new class. Instead of selecting *new Java class* just select *new Java interface*.

Create classes `Book` and `CD`, which implement the Interface. Book has a constructor which is given the author (String), name of the book (String), and the weight of the book (double) as parameters. CD has a constructor which is given the artist (String), name of the CD (String), and the publication year (int). The weight of all CDs is 0.1 kg.

Remember to implement the Interface `Packable` in both of the classes. The classes must work as follows:

```java
public static void main(String[] args) {
    Book book1 = new Book("Fyodor Dostoevsky", "Crime and Punishment", 2);
    Book book2 = new Book("Robert Martin", "Clean Code", 1);
    Book book3 = new Book("Kent Beck", "Test Driven Development", 0.5);

    CD cd1 = new CD("Pink Floyd", "Dark Side of the Moon", 1973);
    CD cd2 = new CD("Wigwam", "Nuclear Nightclub", 1975);
    CD cd3 = new CD("Rendezvous Park", "Closer to Being Here", 2012);

    System.out.println(book1);
    System.out.println(book2);
    System.out.println(book3);
    System.out.println(cd1);
    System.out.println(cd2);
    System.out.println(cd3);
}
```

Prints:

```
Sample output

Fyodor Dostoevsky: Crime and Punishment
Robert Martin: Clean Code
Kent Beck: Test Driven Development
Pink Floyd: Dark Side of the Moon (1973)
Wigwam: Nuclear Nightclub (1975)
Rendezvous Park: Closer to Being Here (2012)
```

NB: The weight is not printed

Box

Make a class called `Box`. Items implementing the `Packable` interface can be packed into a box. The `Box` constructor takes the maximum capacity of the box in kilograms as a parameter. The combined weight of all items in a box cannot be more than the maximum capacity of the box.

Below is an example of using a box:

```java
public static void main(String[] args) {
    Box box = new Box(10);

    box.add(new Book("Fyodor Dostoevsky", "Crime and Punishment", 2)) ;
    box.add(new Book("Robert Martin", "Clean Code", 1));
    box.add(new Book("Kent Beck", "Test Driven Development", 0.7));

    box.add(new CD("Pink Floyd", "Dark Side of the Moon", 1973));
    box.add(new CD("Wigwam", "Nuclear Nightclub", 1975));
    box.add(new CD("Rendezvous Park", "Closer to Being Here", 2012));

    System.out.println(box);
}
```

Prints

```
Sample output

Box: 6 items, total weight 4.0 kg
```

NB: As the weights are saved as a double, the calculations might have some small rounding errors. You don't need to worry about them.

Box weight

If you made an class variable `double weight` in the Box class, replace it with a method which calculates the weight of the box:

```java
public class Box {
    //...

    public double weight() {
        double weight = 0;
        // calculate the total weight of the items in the box
        return weight;
    }
}
```

When you need the weight of the box, for example when adding a new item to the box, you can just call the weight method.

The method could also return the value of an object variable. However here we are practicing a situation, where we do not have to maintain an object variable explicitly, but can calculate its value as needed. After the next exercise storing the weight as an object variable would not necessary work anyway. After completing the exercise have a moment to think why that is.

A Box is packable too!

Implementing the `Packable` Interface requires a class to have the method `double weight()`. We just added this method to the Box class. This means we can make the Box packable as well!

Boxes are objects, which can contain objects implementing the `packable` Interface. Boxes implement this Interface as well. So **a box can contain other boxes!**

Try this out. Make some boxes containing some items, and add some smaller boxes to a bigger box. Try what happens, when you put a box in itself. Why does this happen?

Programming exercise:

Herds (2 parts)

In this exercise we are going to create organisms and herds of organisms that can move around. To represent the locations of the organisms we'll use a **two-dimensional coordinate system**. Each position involves two numbers: `x` and `y` coordinates. The `x`coordinate indicates how far from the origin (i.e. point zero, where x = 0, y = 0) that position is horizontally. The `y` coordinate indicates the distance from the origin vertically. If you are not familiar with using a coordinate system, you can study the basics from Wikipedia.

The exercise base includes the interface `Movable`, which represents something that can be moved from one position to another. The interface includes the method `void move(int dx, int dy)`. The parameter `dx` tells how much the object moves on the x axis, and dy tells the distance on the y axis.

This exercise involves implementing the classes `Organism` and `Herd`, both of which are movable.

Implementing the Organism Class

Create a class called `Organism` that implements the interface `Movable`. An organism should know its own location (as x, y coordinates). The API for the class `Organism` is to be as follows:

- **public Organism(int x, int y)**
  The class constructor that receives the x and y coordinates of the initial position as its parameters.
- **public String toString()**
  Creates and returns a string representation of the organism. That representation should remind the following: `"x: 3; y: 6"`. Notice that a semicolon is used to separate the coordinates.
- **public void move(int dx, int dy)**
  Moves the object by the values it receives as parameters. The `dx` variable contains the change to coordinate `x`, and the `dy` variable ontains the change to the coordinate `y`. For example, if the value of `dx` is 5, the value of the object variable `x` should be incremented by five.

Use the following code snippet to test the `Organism` class.

```java
Organism organism = new Organism(20, 30);
System.out.println(organism);
organism.move(-10, 5);
```

```
System.out.println(organism);
organism.move(50, 20);
System.out.println(organism);
```

Sample output

```
x: 20; y: 30
x: 10; y: 35
x: 60; y: 55
```

Implementing the Herd

Create a class called `Herd` that implements the interface `Movable`. A herd consists of multiple objects that implement the Movable interface. They must be stored in e.g. a list data structure.

The `Herd` class must have the following API.

- **public String toString()**
  Returns a string representation of the positions of the members of the herd, each on its own line.
- **public void addToHerd(Movable movable)**
  Adds an object that implements the `Movable` interface to the herd.
- **public void move(int dx, int dy)**
  Moves the herd with by the amount specified by the parameters. Notice that here you have to move each member of the herd.

Test out your program with the sample code below:

```
Herd herd = new Herd();
herd.addToHerd(new Organism(57, 66));
herd.addToHerd(new Organism(73, 56));
herd.addToHerd(new Organism(46, 52));
herd.addToHerd(new Organism(19, 107));
System.out.println(herd);
```

Sample output

```
x: 73; y: 56
x: 57; y: 66
x: 46; y: 52
x: 19; y: 107
```

Programming exercise:

Animals (4 parts)

In this exercise you'll demonstrate how to use inheritance and interfaces.

Animal

First implement an abstract class called `Animal`. The class should have a constructor that takes the animal's name as a parameter. The Animal class also has non-parameterized methods eat and sleep that return nothing (void), and a non-parameterized method getName that returns the name of the animal.

The sleep method should print "(name) sleeps", and the eat method should print "(name) eats". Here (name) is the name of the animal in question.

Dog

Implement a class called `Dog` that inherits from Animal. Dog should have a parameterized constructor that can be used to name it. The class should also have a non-parameterized constructor, which gives the dog the name "Dog". Another method that Dog must have is the non-parameterized bark, and it should not return any value (void). Like all animals, Dog needs to have the methods eat and sleep.

Below is an example of how the class Dog is expected to work.

```java
Dog dog = new Dog();
dog.bark();
dog.eat();

Dog fido = new Dog("Fido");
fido.bark();
```

Sample output

```
Dog barks
Dog eats
Fido barks
```

Cat

Next to implement is the class `Cat`, that also inherits from the Animal class. Cat should have two constructors: one with a parameter, used to name the cat according to the parameter, and one without parameters, in which case the name is simply "Cat". Another method for Cat is a non-parameterized method called purr that returns no value (void). Cats should be able to eat and sleep like in the first part.

Here's an example of how the class Cat is expected to function:

```
Cat cat = new Cat();
cat.purr();
cat.eat();

Cat garfield = new Cat("Garfield");
garfield.purr();
```

Sample output

```
Cat purrs
Cat eats
Garfield purrs
```

NoiseCapable

Finally, create an interface called `NoiseCapable`. It should define a non-parameterized method makeNoise that returns no value (void). Implement the interface in the classes Dog and Cat. The interface should take use of the bark and purr methods you've defined earlier.

Below is an example of the expected functionality.

```
NoiseCapable dog = new Dog();
dog.makeNoise();

NoiseCapable cat = new Cat("Garfield");
cat.makeNoise();
Cat c = (Cat) cat;
c.purr();
```

Sample output

```
Dog barks
Garfield purrs
Garfield purrs
```