Exercises module 19 – Abstract classes

19.1 – Abstract classes

1. In this exercise, we will improve the code we wrote in the previous exercise and explore how the introduction of **abstract classes** can improve the code.

2. In the previous exercises we worked on the following product classes:

```
class Product {
    public int id;
    public String productName;
}

class Processor extends Product {

    public int frequency;
    public int numberOfCores;
}

class Keyboard extends Product {
    public boolean gotNumericKeypad;
    public String color;
    public boolean isWireless;
}


class Monitor extends Product {
    public int size;
    public float weight;
}
```

3. These classes we can initialize using code like:

```
Processor cpu1 = new Processor();
Processor cpu2 = new Processor();
Keyboard kbd = new Keyboard();
Monitor monitor = new Monitor();

cpu1.productName = "Intel Core I7";
cpu1.id = 1;
cpu1.frequency = 3000;
cpu2.productName = "Amd threadripper";
cpu2.id = 2;
cpu2.frequency = 3500;
kbd.productName = "Microsoft Natural keyboard";
kbd.id = 3;
monitor.productName = "Samsung T191T";
monitor.id = 4;
```

4. Make the Product class abstract and add an abstract void method named **Describe()**

5. Implement the necessary code in the sub-classes to make it compile. In the methods print out all the product details, so that for example for the CPU we get a print-out like:

```
Name: Intel Core I7 (ID:1)
 frequency: 3000
 numberOfCores: 0
```

6. Add the four products to an **ArrayList** like we did in the previous exercise and pass it to a **printProducts** method that will call the Describe method on each item in the list.

   The desired output should be something like:

```
Name: Intel Core I7 (ID:1)
 frequency: 3000
 numberOfCores: 0
Name: Amd threadripper (ID:2)
 frequency: 3500
 numberOfCores: 0
Name: Microsoft Natural keyboard (ID:3)
 gotNumericKeypad: false
 color: null
 isWireless: false
Name: Samsung T191T (ID:4)
 size: 0
 weight: 0.0
```

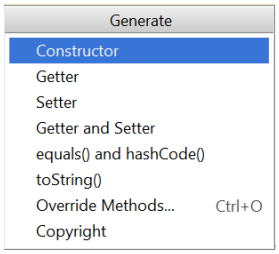
19.2 – Drawing program (Advanced)

1. Create a new project and import the **com.googlecode.lanterna** library (V2.1.9)

2. Create a class named **Point**, like this:

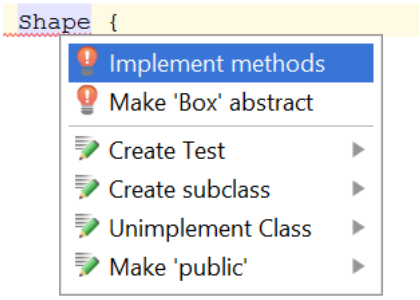
```
public class Point
{
    public int x;
    public int y;
}
```

3. Then we need to create a constructor for this class. We let **IntelliJ** create one for us by placing the cursor inside the class and then press **Alt+Insert** and select **Constructor**



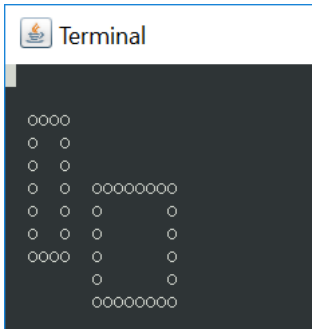
(Then select the two fields and press OK)

4. Create a new **abstract** class named **Shape** that contains a simple abstract method named **List<Point> draw();**
5. Create a new class named **Rectangle** that **inherits** from the abstract **Shape** class
6. Place the cursor on the Shape class word and press **Alt+Enter** and **implement the method**



7. Add two private fields to this class named **start** and **end** of type **Point** (That we created earlier).
8. Create a **constructor** that sets the **start & end** fields (using alt+Insert)
9. Now we need to implement the draw function. To make this application "simple" we let the draw method return a list of coordinates to draw on the screen.

1. Create a new empty List of Points and then add the code to return a List of coordinates for drawing a box. The coordinates are later passed to a drawing method. Given a pair of **start,end** coordinates, the goal of the code is to draw images like:



10. The code for the main method should be something similar to:

```
Shape shape1 = new Rectangle(new Point(8,5), new Point (15,10));
Shape shape2 = new Rectangle(new Point(2,2), new Point (5,8));

List<Shape> shapes = new ArrayList<>();
shapes.add(shape1);
shapes.add(shape2);

UI gui = new UI(terminal);
gui.drawShapes(shapes);
```

11. Create the missing **UI** class that takes a list of shapes and draw it on the screen.

    Basically it should for each shape in the provided list, call the draw method to get the coordinates to draw.

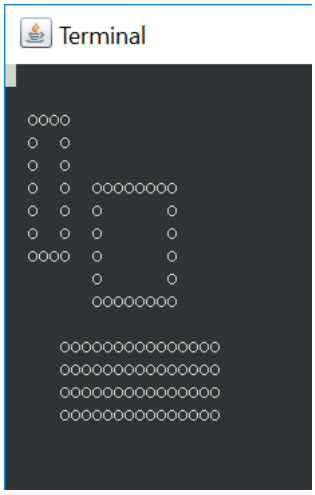
Extending the drawing application

12. Extend the system with a class **FilledRectangle** (copy the code in the Rectangle class and change the draw code logic).

13. Add the FilledRectangle to the list of shapes, like:

```
Shape shape1 = new Rectangle(new Point(8,5), new Point (15,10));
Shape shape2 = new Rectangle(new Point(2,2), new Point (5,8));
Shape shape3 = new FilledRectangle(new Point(5,12), new Point (20,16));

List<Shape> shapes = new ArrayList<>();
shapes.add(shape1);
shapes.add(shape2);
shapes.add(shape3);
```

Running the application could generate an image like:

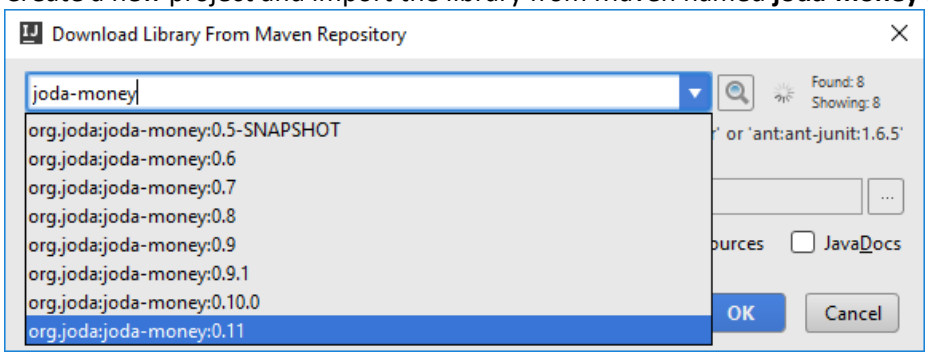


If you have time:

- If you have time you could extend this code by adding shapes like Dot, VerticalLine, HorizontalLine, Line, Circle …
- Perhaps add animation and let the shapes move on the screen? Zoom? Colors?

19.3 – Overriding

1. In this exercise, we are going to explore the use of overriding **toString** and how to handle money.
2. Create a new project and import the library from Maven named **joda-money (0.11)**



3. We will use **joda-money** as the library to represent money in our application that we are going to create. Do visit http://www.joda.org/joda-money/ and review how it is used.

4. The end goal of this application is to write code like this:

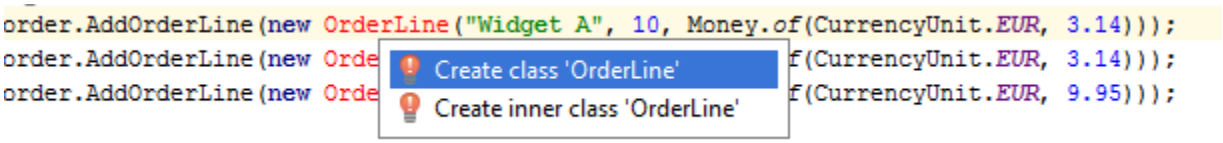
```
Order order = new Order("Edument AB");

order.AddOrderLine(new OrderLine("Widget A", 10, Money.of(CurrencyUnit.EUR, 3.14)));
order.AddOrderLine(new OrderLine("Widget B", 10, Money.of(CurrencyUnit.EUR, 9.95)));

System.out.println(order);

System.out.println("Total: " + order.GetOrderTotal());
```

Running the application should result in an output like:

```
Edument AB
  Widget A, 10 items, price per item EUR 3.14
  Widget B, 10 items, price per item EUR 9.95
Total: EUR 130.90
```

(We only support EUR as currency)

5. Start by copy or type the code above into your main method and then press **alt+enter** to create the **Order** and **OrderLine** classes



6. For the **OrderLine** class
   a. Create these public **fields** and a **constructor** that sets them.

```
public String productName;
public int quantity;
public Money price;
```

b. Override the **toString** method so that it returns a string like:

```
Widget A, 10 items, price per item EUR 9.95
```

c. Create a method named **GetOrderLineTotalValue()** that returns the total value of the orderline object. The return type should be money. Use the **multipliedBy** method.

7. For the **Order** class
   a. Create these two **fields:**

```
private List<OrderLine> orderlines;
public String customerName;
```

b. Create a **constructor** that sets the CustomerName as an input parameter and initializes the orderlines with a new empty list.

c. Create an **addOrderLine** method that take an **orderline** as input parameter and returns **void**. It should add the **orderline** to the orderlines field.

d. Create a **getAllOrderLines()** method that returns a list of orderlines.

e. Override **toString** that returns a string with the output like:

```
Edument AB
  Widget A, 10 items, price per item EUR 3.14
  Widget B, 10 items, price per item EUR 9.95
```

(It should call the **toString** on each **orderline** to get the text for each line.)

f. Add a method named **getOrderTotal**() with Money as the return type and that returns the sum of all the orderlines value.

8. Run the application and you should get an output as shown

```
Edument AB
  Widget A, 10 items, price per item EUR 3.14
  Widget B, 10 items, price per item EUR 9.95
Total: EUR 130.90
```

Questions and concepts to study further on your own:

- The SOLID Principles
  https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
- Abstract classes
- Can you create an instance of an abstract class?
- What is the benefit of writing abstract methods in an abstract class?

.